CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER’S THESIS

Title:	Semi-Supervised Learning of Millions of Astronomical Spectra
Student:	Bc. Andrej Palička
Supervisor:	RNDr. Petr Škoda, CSc.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

The current archive of the LAMOST telescope contains about 4 million of astronomical spectra. The goal of the thesis is to identify objects with interesting spectral line profiles (namely emission line stars) based on a small set of examples.

1) Make a survey of semi-supervised training methods and define the optimal strategy of learning, taking into consideration massively parallel solutions (e.g., SPARK, GPUs ...).
2) Select spectra of known emission-line stars from archives of the Ondřejov 2m telescope.
3) Make a positional cross-match of objects from 2) with LAMOST survey to get a sample for training.
4) If the sample obtained in 3) is insufficient, simulate the expected appearance of spectra from 2) in the LAMOST spectrograph to get a training sample.
5) Identify new objects of the same class in the whole LAMOST survey and make visual confirmation.
6) Discuss the success rate and computing performance of your strategy.
Try to integrate your solution into the system VO-CLOUD.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 19, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Science

Master's thesis

Semi-Supervised Learning of Millions of Astronomical Spectra

Bc. Andrej Palička

Supervisor: RNDr. Petr Škoda, CSc.

9th May 2016

Acknowledgements

I want to thank my supervisor, RNDr. Petr Škoda, for his guidance and support.
I also want to thank Ing. Ivan Šimeček, PhD. for accepting my request to be my reviewer.
Last but not least, I want to thank my family and friends for supporting me throught the whole ordeal. I could not have done it without you.
This research made use of the following libraries from the Scipy ecosystem: Scipy, Numpy and Matplotlib.
This research made use of Astropy, a community-developed core Python package for Astronomy (Astropy Collaboration, 2013).
Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated. Specifically, I would like to thank Mr. Petr Hanousek and Mr. František Dvořák for their help with the Spark cluster.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 9th May 2016 . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Palička, Andrej. *Semi-Supervised Learning of Millions of Astronomical Spectra.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Použili sme čiastočne riadené učenie na detekciu emisných spektier v archíve z observatória LAMOST za pomoci masívne paralelného prostredia Spark. Implementovali sme aplikáciu, ktorá tieto spektrá predspracuje a aplikuje sériu transformácii aby sme tieto dáta mohli použiť na trénovanie modelov. Ďalej sme implementovali algoritmy čiastočne riadeného učenia, založené na grafovej reprezentácii dát, zvané Label Propagation a Label Spreading. tieto algoritmy používame na naučenie modelu, ktorý spektrá bude klasifikovať. Aplikovali sme tieto algoritmy na podmnožinu archívu, ktorej veľkosť bola jeden milión spektier.

Klíčová slova strojové učenie, čiastočne riadené učenie, astroinformatika, hviezdy s emisnou krivkou, LAMOST

Abstract

We use semi-supervised learning to detect spectra with emission in an archive from the LAMOST observatory using a massively parallel environment called Spark. We have implemented a preprocessing application that would take original raw spectra and apply series of transformations in order for them to be usable for training models. We have also implemented graph-based semi-supervised algorithms Label Propagation and Label Spreading. We use these to fit the models and then classify the spectra. We have applied these algorithms to a subsample of the archive of size one million of spectra.

Keywords machine learning, semi-supervised learning, astroinformatics, emission-line spectra, LAMOST

Contents

List of Figures

List of Tables

Introduction

We live in an era when every imaginable area of human endeavour produces and consumes huge amount of data. This data need to be stored, processed and most important, presented in a way we can use to make decisions, gain insight into some problem or discover new knowledge.

Algorithms and systems have been developed to help with all of the aforementioned problems. We now have distributed file systems, such as *HDFS*, distributed databases, such as *Hbase* or *Cassandra*. We can process large amount of data using paradigms such as *MapReduce*. Libraries such as *Sparks' MLlib* help discover relationship between the data and create models on top of them.

This diploma thesis concerns itself primarily with the machine learning part of knowledge engineering. We use it to process and classify large amounts of stellar spectra in an automated way.

Machine Learning

Machine learning is a field of computer science whose primary aim is to design algorithms, that allow computers to learn patterns and relationships in data without the need for a human to explicitly define this knowledge. They build models that attempt to capture some knowledge hidden in the data. This includes classifying data into some predetermined classes, detecting outliers, finding clusters or transforming the features of the data to be more useful for subsequent machine learning tasks.

There are two main classes of machine learning algorithms:

Supervised These algorithms have some prior knowledge about the data, that was supplied by some external means. This is usually done by a human domain expert. Main representatives are the *classification* tasks, where the models classify data into some predefined, finite set of classes and *regression* tasks, which infer output of a real-valued function.

An example of a regression task would be a model, that would infer a temperature based on features such as time of a day, humidity and cloudiness. An example of a classification is a task that would simply say if it is cold, mellow or hot outside.

Unsupervised These algorithms do not have any prior knowledge of the data. It attempts to discover these relationships itself. This is by definition a much harder task than supervised learning, however it can potentially lead to much more interesting results. Example tasks are clustering, automated feature selection or outlier detection.

These classes do have some specific subclasses that group similarly-working algorithms together. One particular subclass is a *semi-supervised learning*, which we further explore in this thesis. It is a group of algorithms designed to work on datasets, that have very few labelled data points compared to the amount of unlabelled points.

Astroinformatics

Astroinformatics is an interdisciplinary field combining astronomy and informatics. Its' main purpose is to devise new ways on how to store the scientific data from astronomical measurements, transform them and present them to both scientists and general public [1]. The greatest representative of this field and sort-of-a umbrella project is the Virtual Observatory [2].

Virtual Observatory is a collection of various databases, tools and protocols. It attempts to unify the various data formats, protocols and processes so that astronomers and astronomical institutions and societies can more easily collaborate and access each other's data.

Observatory in Ondřejov is one such institution. They have build a VO-Cloud [3], a system for running data processing jobs on archives of stellar spectra. They expose their archives through a VO-based Simple Spectral Access Protocol, which allows an easy access to spectra. One of their goals is to identify *Be stars* and their types. VOCloud already has some modules, that allow for building models that could classify specra into these types, however they are not designed for large archives with millions of spectra. This was not an issue initially, however they have managed to obtain data from a newly-built Chinese observatory LAMOST. These archives contain hundreds of gigabytes of spectra, totalling to several millions of individual measurements.

Therefore the goal of this thesis is to design an application that could deal with large amount of data and that would scale with the increase of size. One of the specifics of our problem is that we have a small training set that was already labelled for our previous experiments, however the dataset we need to classify is much, much larger. For this we have decided to explore the field of semi-supervised learning. The algorithms of this type are designed to deal with exactly this issue. Specifically, we implement and use distributed versions of two graph-based algorithms, called Label Propagation and Label Spreading.

We also implement a data processing module, that somewhat mirrors some of the capabilities of existing Ondřejov subsystems, such as VOCloud and spectra retrieval service, Datalink. The motivation for this was that these systems were not designed for such amount of data. Our module not only parses the spectra in FITS or VOTABLE data format, it also performs a rebinning into wavelength scale, cuts the spectra at user-specified wavelengths, resamples them in case they come from different sources and ultimately stores them in a format that is usable for subsequent analysis.

To deal with the amount of data we went with state-of-the-art technologies. We use *Spark* as our distributed engine. It improves on the traditional *MapReduce* algorithm in several ways, which are described later in the thesis. Nevertheless, we still use a file system built on *Hadoop*, *HDFS* to store our data.

CHAPTER **1**

Semi-Supervised Learning

In this chapter, we shall explore the theory behind semi-supervised learning and make a survey of commonly used algorithms. We shall begin with a brief description of supervised and unsupervised learning, so that we may better understand, how semi-supervised learning builds on top of these.

1.1 Supervised and Unsupervised Learning

Supervised learning is a class of machine learning algorithms, where we have a priori information about the nature of our data. Let us have a space $\mathcal{X}$ from which we sample n vectors, producing a sequence $X = (x_i), i \leq 0 \leq n$. Let us also define sequence of labels, sampled from $\mathcal{Y}$, $Y = (y_i), 0 \leq i \leq n$. Supervised learning then attempts to estimate a distribution of $p(y|x)$. If $\mathcal{Y}$ is a discrete space we say we perform *classification*, because we are assigning classes to vectors. If it is continuous, we are doing *regression*, because we are estimating an output of a continuous function. Supervised learning builds the model by minimising the error of the output.

Unsupervised learning, on the other hand, has no prior information about the structure of the data. It operates only with the information from the X itself. The goal of unsupervised learning is to estimate the density of $\mathcal{X}$, which is not always feasible. Therefore, we usually attempt to achieve simpler goals, such as *clustering*, *outlier detection* or *feature selection.*

1.2 Semi-Supervised Learning

Let $X = (x_0, \ldots, x_n)$ be our dataset. Then $X_l = (x_0, \ldots, x_m)$ and $X_u = (x_{m+1}, \ldots, x_n)$ be sequences of identically-sized vectors sampled from $\mathcal{X}$. Then let $Y = (y_0, \ldots, y_m)$ be a set of known labels and $Y = (y_{m+1}, \ldots, y_n)$ a set of unknown labels, where y_i is a label for x_i. This is how semi-supervised data set *usually* looks like. This resembles a supervised learning in that we have labelled data, however with an addition of also having unlabelled data, which may help us estimate the distribution of the data set more precisely.

There is, however, another possible variant of semi-supervised learning. Here, we are not using labelled data, but merely some constraints. These constraints may link some points, that share the same label, or they may reveal the actual number of classes. This resembles unsupervised learning, however with some a priori information about the data.

We may also differentiate between semi-supervised algorithms by the goal they are trying to achieve. *Transducive* methods seek to only label the unlabelled data X_u. *Inductive* methods, on the other hand, attempts to find a mapping $f : \mathcal{X} \to \mathcal{Y}$, that predicts a class to any point x from $\mathcal{X}$.

So how exactly may semi-supervised learning help us? If we want semi-supervised learning to bring us any improvement over supervised methods, we need to make sure that the information about the distribution of the data that X_u carries, $p(x)$ is useful in inferring the labels, $p(y|x)$. Several assumptions should be met, for semi-supervised learning to work.

1.3 Assumptions for Semi-Supervised Learning

1.3.1 The smoothness assumption

The *smoothness assumption* states that if two points x_i and x_j are close in a high-density region, then their respective outputs y_i and y_j should also be close [4].

If this assumption would not hold, our data would resemble a random mess, where it would be impossible to find a general decision boundary.

This assumption is a slight modification of a rule from supervised learning, which considers only the distance between two points. It assumes that the target value smoothly changes with the distance. This is sufficient for supervised learning, because we have complete information about the training data. With semi-supervised learning, we also need to consider the density of a region, because it allows us to make assumptions on data for which we have no target value.

This applies for both classification, and for regression. For classification it simply means that their classes are *most likely* the same, whereas in regression it would mean that the outputs are close relative to the function that generated them.

1.3.2 The cluster assumption

The *cluster assumption* states: If two data points x_i and x_j belong to the same cluster, then they likely share the same class.

Let's say we have run a clustering algorithm on our training data and have discovered, that the data are neatly organized in clusters. We then might look at the majority of labelled data in each cluster and proclaim, that each unlabelled data point in that cluster shares the same class. This would of course be a naive approach, but it has been used in some early SSL algorithms.

This assumption might seem like a special version of the smoothness assumption from Section 1.3.1. It can be formulated as a *low-density separation assumption*: The decision boundary lies in a low-density region [4].

To illustrate how it connects to the the smoothness assumption consider this. The decision boundary that lies in the low-density region separates different clusters (high-density regions). We assume that data points in these high-density regions share the same class and thus the assumption still stands. If however this was not the case and we would put a decision boundary through a cluster, it would divide the points in that cluster into two different classes. This would break the first assumption, since there would be points that are close to each other, but that were classified as a different class.

1.3.3 The manifold assumption

The manifold assumption states that the (high-dimensional) data lie on a low-dimensional manifold [4]. This assumption helps to deal with the so-called curse of dimensionality. With the growing number of dimensions, the volume of space and amount of data needed for sound statistical analysis grow exponentially. Algorithms then have a problem with estimating densities of each dimension or with computing pairwise distances between data points, as they become less clear with more dimensions.

By assuming that even though the data are high-dimensional but they lie on some lower-dimensional manifold, they can operate in the lower-dimensional space.

1.3.4 Transduction

Transduction is based on Vapnik's principle. It states that when we try to solve a problem, we should not attempt to solve a more difficult problem in the process. In the domain of machine learning it means, that when we try to find labels for unlabelled points X_u, we should not try to estimate the entire density of $\mathcal{X}$ in order to do so.

Inductive learning indeed does estimate the whole probability density, trying to find a function $f_i : \mathcal{X} \to \mathcal{Y}$ that would work generally for all x. Transduction, on the other hands, only seeks to find a function $f_t : X_u \to \mathcal{Y}$.

1.4 Classes of Semi-Supervised Learning

There are multiple ways of classifying the semi-supervised models. We have chosen the taxonomy according to [4], as it models quite closely how different algorithms actually work.

1.4.1 Graph-Based Models

Let $G = (V, E)$ be a graph with weighted edges representing our data. Nodes represent the data points and the weights of edges the pairwise similarities between the neighbouring data points. Since the data need not be a graph, what constitutes neighbourship is open to discussion and is left for the concrete implementation to solve. One of the simplest methods would be to to apply *K-nearest neighbours* on the data and use those neighbours for incidence.

Now let $w : V \times V - > R$ be the similarity function, which takes two data points represented as nodes and computes their similarity. Similarity should be a positive, symmetric function. Then let us define an adjacency matrix $W \in R^{|V|,|V|}$:

$$W_{i,j} = \begin{cases} w(e) & \textit{if } e = (i, j) \in E \\ 0 & \textit{otherwise} \end{cases}$$

1.4.1.1 Label Propagation

Label propagation is an algorithm that leverages the graph representation of data to fit a model. It is a transductive algorithm, however an inductive version can be derived. Labels are encoded as a one-hot variable, so that we may support multi-class classification. The algorithm basically computes weights for labels for each data point based on the distance to its' neighbours. Note that in this version of the algorithm, the initial labels do not change and are reset to their original value in each step, however this may vary in the implementation. A basic pseudocode is provided in Algorithm 1:

Algorithm 1: Label Propagation

```
Data: $W$: weighted adjacency matrix, $Y : (y_0, \dots, y_m, \underbrace{0, \dots, 0}_{n-m})$
Result: $\hat{Y}$: label weights for each input point
begin
    compute a degree matrix $D_{i,i} \leftarrow \sum_{j=0}^{n} W_{i,j}$
    construct a probabilistic transition matrix $P \leftarrow D^{-1}W$
    $\hat{Y}^{(0)} \leftarrow Y$
    while $\hat{Y}$ not converged to $\hat{Y}^{(\infty)}$ do
        $\hat{Y}^{(i+1)} \leftarrow P\hat{Y}^{(i)}$, the matrix multiplication is computed for each
        part of one-hot variable separately
        set the labelled part of $\hat{Y}^{(i+1)}$ back to the original values
    return $\hat{Y}^{(\infty)}$
```

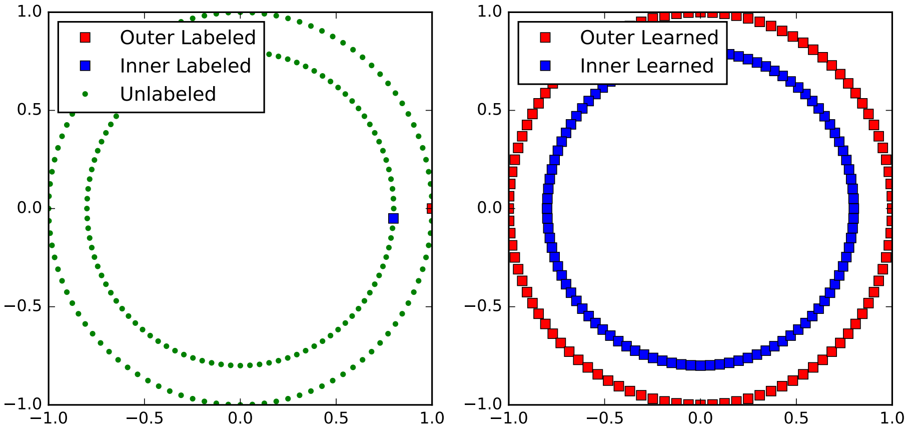


Figure 1.1: Label Spreading visualisation [5]

1.4.1.2 Label Spreading

A similar algorithm to label propagation is the label spreading. This algorithm, uses the normalized graph Laplacian $\mathcal{L}$ to propagate the label information across the graph. It also allows the labels to retain some partial information from the initial labelling. The pseudocode is shown in Algorithm 2.

Algorithm 2: Label Spreading

```
Data: $W$: weighted adjacency matrix, $W_{i,i} = 0$;
$Y : (y_0, \dots, y_m, \underbrace{0, \dots, 0}_{n-m})$;
$\alpha$: $alpha \in [0, 1)$ a ratio of how much the Laplacean and original
labelling will influence the result
Result: $\hat{Y}$: label weights for each input point
begin
    $D_{i,i} \leftarrow \sum_{j=0}^{n} W_{i,j}$
    $\mathcal{L} \leftarrow D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$
    $\hat{Y}^{(0)} \leftarrow Y$
    while $\hat{Y}$ not converged to $\hat{Y}^{(\infty)}$ do
        $\hat{Y}^{(i+1)} \leftarrow \alpha \mathcal{L} \hat{Y}^{(i)} + (1 - \alpha) \hat{Y}^{(0)}$
    return $\hat{Y}^{(\infty)}$
```

1.4.1.3 Inductive setting

The algorithms above clearly work as transductive algorithms, since they do not build a general model. However it is possible to extend them to be able to classify unseen examples as well.

Say we receive a new point, x. We can then use the same similarity function w, that we have used in the initial graph construction, to find its' adjacent points. They we weight the label of each adjacent point by the distance and then simply normalise the result by the sum of the distances. Notice, that if we are using k-neighbours as our similarity function, then the induction is the same as KNN classification. This yields the following formula:

$$\hat{y} = \frac{\sum_j w(x, x_j) y_j}{\sum_j w(x, x_j)}$$

1.4.2 Low-Density Separation Models

In low-density separation, we attempt to place a decision boundary in an area of low density. Thus it is directly implementing the low-density separation assumption (Section 1.3.2).

A classic algorithm is the Transductive Support Vector Machine. Unlike the standard SVM, which maximises the margin of hyperplanes with regards only to the labelled data, the TSVM maximises the margin relative to all the data, both labelled and unlabelled.

Let's consider $\mathcal{Y} = \{-1, 1\}$, therefore each y_i may only become 1 or -1. We shall later show how it is possible to generalise this model for multiple classes. We are searching for two parallel hyperplanes $h_1 : wx + b = 1$ and $h_{-1} : wx + b = -1$ with the greatest distance $\frac{2}{||w||}$ between them, which is the margin. The decision boundary then lies in the middle between these two hyperplanes, its' formula being $h : wx + b = 0$. Hyperplanes h_1 and h_{-1} are called *supporting hyperplanes* and any vectors x that satisfy their equation are called *supporting vectors*. Note that in constrast with the standard SVM, where x is taken only from the labelled data X_l, in the TSVM we maximise the margin over all the data X.

1.4.2.1 Hard-margin

When the data are linearly separable, this leads to a nicely defined decision boundary. The following should hold for labels y_i:

$$\begin{cases} 1 & \text{for } wx + b \geq 1 \\ -1 & \text{for } wx + b < 1 \end{cases}$$

This is called a hard-margin problem, as there is no ambiguity about where the margin should be.

In terms of an optimisation problem, it could be written as:

$$\begin{aligned}
\text{min} \qquad & SVM(y_{m+1}, \dots, y_n, w, b) = \frac{||w||}{2} \\
\text{s. t.} \qquad & \forall i, 0 \leq i \leq m : y_i \left(wx_i + b \right) \geq 1 \\
& \forall i, m+1 \leq i \leq n : y_i \left(wx_i + b \right) \geq 1 \\
& \forall i, m+1 \leq i \leq n : y_i \in \{-1, 1\}
\end{aligned}$$

Therefore, we are looking for w, b and for labels of the unlabelled data so with the maximal margin. Here, the cluster assumption really comes into play, as we assume that the clusters in the data correspond to the labels. This assumption allows us to also use the unlabelled data for finding the decision boundary.

1.4.2.2 Soft-margin

However in reality, data are rarely linearly separable and neatly organised in clusters. The usually contain outliers, or the low-density space disappears in some places.

There are several ways of dealing with this. We can introduce a slack variable, also called a hinge loss. This variable sets the tolerance of SVM to having data points with a particular label on the wrong side of the boundary.

1.4.2.3 Kernel trick

Another option would be to use kernel trick. Kernel trick transforms the input data into a higher dimensional space, where the data may be linearly separable.

Note, that despite its' name, the TSVM can also be used as an inductive model by using it as a standard SVM.For any additional data points we may want to classify, that were not part of the original unlabelled data, we compute on which side of the decision boundary the data point belongs.

1.4.3 Generative Models

Generative models attempt to estimate the joint probability distribution $P(x, y)$. This allows the model to generate data points from this distribution, but it also serves as an intermediate step for computing the conditional distribution of $P(y|x)$. This distribution can be computed using the Bayes theorem.[6]

To model the joint probability $P(x, y)$, the algorithms actually has to estimate the conditional probability $P(x|y)$. This is done by estimating over $P(x|y, \theta)$, where θ represents the paramters of the model. Similarly we need to estimate $P(y)$ by modelling $P(y|\pi)$.

$$P(y|x) = \frac{P(y) * P(x|y)}{\sum_{y}^{\mathcal{Y}}(P(x|y_i) * P(y_i))}$$

Then to get the model for the marginal $P(x)$:

$$P(x) = \sum_{y} P(y)P(x|y,\theta)$$

The problem with the generative models is that they are trying to estimate the distribution behind X, which is generally nontrivial. Instead of trying to directly estimate $P(y|x)$, the algorithms in this class waste resources on estimating $P(x,y)$, which we may not ultimately need if we do not wish to be able to generate data points.

An example of such algorithm is a expectation–maximization algorithm, which attempts to find parameters of a given model. We assume this model is the one that generated our training and testing set.

Recently [7] a new algorithm was proposed that uses deep learning neural networks to estimate these parameters. These networks can discover hidden relationships in the data very well thanks to having multiple layers of neurons. However their drawback is that they require a lot of training data and the training process is quite resource-hungry.

1.4.4 Change of Representation models

Change of representation methods modify the input space. They operate in two steps:

1. Perform an unsupervised step. Here we change the representation of the data, modify the distances or metrics or apply similar transformation to the data.

2. Perform a supervised step using leveraging the change in the first step. This is usually a semi-supervised algorithm for which it was needed to transform the input data.

These algorithms are often an extension or modification of algorithms from previous classes. They implement the smoothness assumption, because they are trying to enhance the small distances in high-density regions.

1.4.5 Self-training scheme

Self-training scheme is simplest approach to semi-supervised learning. Here we train a model using some standard classification algorithm, such as decision trees, or SVM on only the labeled data. Then we classify the unlabeled data and gather the predicted labels along with whatever confidence the algorithm uses for choosing the labels, be it a probability, likelihood or any other metric. Then we take a number of the most highly rated newly labeled samples and add them to the training set. Then we iterate the process with the new training set.

Chapter 2

Massively Parallel Environments

In this chapter, we shall describe the implementation part of the thesis. We shall also describe the working environment and techniques paradigmes we have used for the implementation.

2.1 MapReduce paradigm

MapReduce [8] is a distributed design pattern. It is loosely inspired by elementary functional programming functions *Map* and *Reduce*, although the semantics and the functionality slightly differs. *Map* accepts a key for which it returns a value. Keys and their respective values are then sent to the Reduce function as a tuple, which processes the result.

Many distributed jobs follow the same principle: take a large dataset and apply some transformations to it producing a derived dataset. Both input and output dataset are basically a collection of key-value pairs. Many of these computations can be divided into three main steps:

Map Map function is implemented by the user. It takes a K-V pair and applies a transformation to it. Return a intermediary K-V result. The MapReduce engine will group the data under the same key.

Shuffle Shuffle moves the values belonging to the same key. Simple implementations of shuffle may move the values belonging to the same key to only one machine while more complex may further distribute the data to more machines, keeping track of what is where. The shuffle is generally the responsibility of the engine.

Reduce Reduce accepts a key and a collection of values belonging to the same key. Generally an aggregation is then performed on these values, producing a single value that is then returned for this key.

This is the foundation of a MapReduce paradigm. A master node orchestrates the whole cluster. It takes care of data partitioning and tracks what is

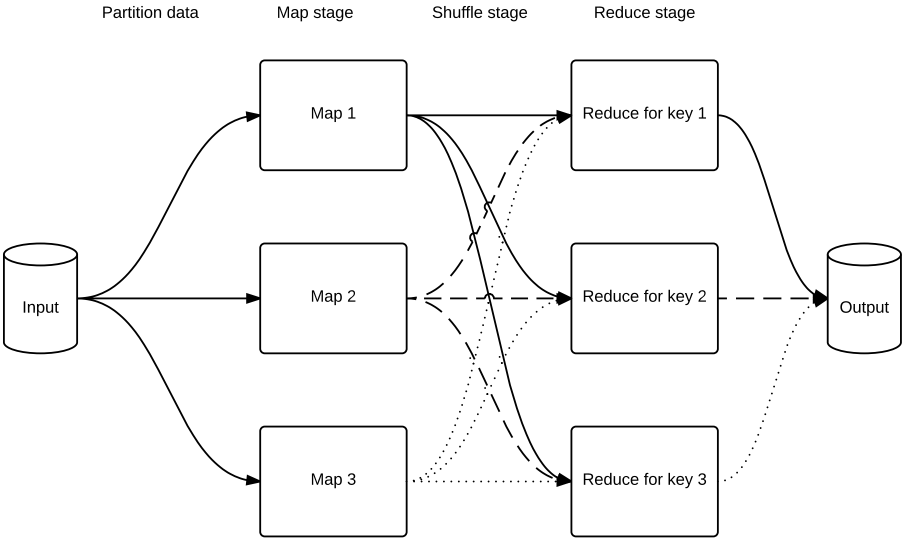


Figure 2.1: MapReduce architecture.

where. Formally, we recognise the mapper and reducer machines which run their respective part. In practice however, the same machine may be used for both Map and Reduce part. In fact, it is desirable for the master node to shuffle the data in such a way that they need not move far from their original machine, ideally staying on the same place.

From the above description it is clear, that MapReduce is great for problems that require large-scale transformations and aggregations of data, where the data can be trivially separated into partitions and shipped to different machines. Typical MapReduce jobs are the ones that compute various metrics over a dataset or query the dataset in batches.

MapReduce was also designed with reliability and robustness in mind. It is meant to run on commodity hardware as opposed to supercomputers. The results of all the intermediate computations are supposed to be either replicated to multiple executor machines or stored on a distributed file system, so that in case of failure of an executor machine the results can be retrieved and computation restarted.

The main bottleneck of MapReduce is usually the shuffle part [9, 10]. This is because the data need to be sent over the network to the appropriate machines and often stored in a persistent location, both for greater redundancy and because they may not fit into the memory. The Map and Reduce parts can also slow down the computation however, particularly when the values are distributed among the different keys in a highly unequal way.

MapReduce in its' basic form is not well suited for iterative algorithms or when we need to repeatedly query the dataset. It also awkward to simulate state in Map and Reduce operations.

2.1.1 Hadoop

One of the most commonly used implementations of MapReduce is Apache Hadoop. It implements the MapReduce paradigm and exposes the API for Java with bindings for other languages. It has these main components:

Hadoop Common This is a collection of common libraries that wrap the functionality used in all the other components.

Hadoop MapReduce This is the MapReduce implementation itself. It is composed of the execution engine as well as of the API's that expose the MapReduce functionality.

YARN The cluster manager. It allocates the resources to jobs, allows users to submit their application and schedules them.

HDFS The distributed file system. It is tightly coupled with MapReduce component as it uses it to load and store results.

While we do not directly use the MapReduce capabilities of Hadoop, we do use HDFS as our data store. It is a fault-tolerant, distributed file system. A HDFS cluster is composed of a single *NameNode* and multiple *DataNodes* [11]. A NameNode acts as a master server, keeping the metadata. The metadata contain information about what files are stored on the filesystem and where they are stored. NameNode also acts as a frontend, accepting the commands from the user.

The files in HDFS are stored on DataNode, split into large blocks, usually of size 64 or 128 MB. Each block is replicated amongst multiple DataNodes. The DataNodes perform operations such as read or write on these blocks when instructed by the *NameNode*.

The fact that HDFS splits the files into fairly large chunks is great when one needs to store a huge file, however storing a large amount of small files means that they will consume far more space than necessary. A common solution is to create a `SequenceFile`, which is a file of key-value pairs, where the key is a name of the original file and the value is its' content.

Appart from HDFS we use YARN for submitting our jobs and for managing the resources. There are several types of agents in a YARN cluster [12]:

ResourceManager There is one RM in the whole cluster. It keeps track of resources across the whole cluster and schedules. It acts as a client for the user.

NodeManager The node manager is responsible for containers running on the machine it is assigned to. It reports back to the ResourceManager.

ApplicationMaster The ApplicationMaster is a per-application agent. It keeps track of what resources the application needs and requests them from the ResourceManager. It also works in concert with the NodeManager to which it reports the state of the job.

The ResourceManager is further composed of two components:

Scheduler The scheduler assigns the resources to applications based on their requirements. It performs no tracking of the application status and it does not offer any redundancy in case of a job fails. It only takes into account the declared requirements of each application.

ApplicationManager The application manager accepts submissions from the the user. It determines where the ApplicationMaster will run and also tracks the state of the execution.

2.2 Spark

Spark [13] is a distributed computing framework. It supports MapReduce paradigm, but improves upon the strictly linear execution plan of the original design. The basic building block is a Resilient Distributed Dataset (RDD). RDD is a distributed collection, which is designed to be fault-tolerant and supports many common functions applicable to collections, such as map, filter or reduce.

2.2.1 RDD

RDD is created either by parallelising and existing local collection, or by reading the data from some sort of data source, such as HDFS, S3 or a database. It is also possible to parallelize a local file, this however must exist on all the computing nodes. Creating an RDD in essence means partitioning the data and assigning each partition to an executor node.

RDDs support two types of operations: *transformations* and *actions*. Transformations are operations that act independetly on each element of the RDD and return a new RDD. Such operations are for example *map*, which processes the input value and outputs something else, or filter which removes those elements from the collections, that do not satisfy a user-defined predicate. Transformations can be chained and are executed in a lazy fashion.

Actions, on the other hand, are operations that aggregate the values in the RDD and return a non-distributed, local result. Such operations are for example *reduce* or *collect*.

By default, intermediary RDDs are not persisted, but instead, when an action is applied on an RDD, all the transformations in the chain are performed. This helps save memory and improves the resiliency of the computations in case there was a node failure. However, if there were repeated actions performed on the dataset, i. e. some sort of iterative algorithm, this would lead to a performance degradation, as the transformations would need to be applied in each iteration. For this reason, RDDs can be persisted in the memory or on a filesystem, which forces the computation of the transformations and caches the final result. Note, that the chain of the transformations is still kept, so that the RDD could be recomputed in case of a failure.

An RDD may either be an unordered [1] collection of objects, kind of like a multiset, or organised by a key. What operations are then available depends on this, for example an RDD without a key does not support operations such as aggregation by key. There are also some specialisations available based on the type in the collection, e. g. and RDD with a numerical type has a function for computing a sum of the values.

2.2.2 DataFrame

A higher-level abstraction over RDDs is a structure called DataFrame. It is a table-like structure, where data are organised in rows and columns. It is not unlike a table in relational databases, in that it has a pre-set schema and supports various SQL-like commands. This allows Spark to somewhat optimise the execution plans when executing operations on them.

A DataFrame may be created either directly from an RDD or from some sort of data source. This may be for example a database, XML file, CSV file or similar structured sources.

A disadvantage of DataFrames from a language point of view is that they are not statically typed as oposed to RDDs. Even though they do store information about they schema, it is not readily available and inferable just from the type of a variable that refers to them. This makes them awkward to work with when one is unfamiliar with their structure.

2.2.3 Execution model

A Spark application runs sequentially in a driver, until a parallel operation is executed. Upon discovering such operation, Spark determines which variables and methods from outer outer scope the operation needs and constructs a closure over them. Computation itself is triggered once an *action* is reached, or if the driver explicitly requests caching. Then a *job* is created and executed in a distributed manner. Each job is comprised of several *tasks*.

[1] Although the RDD may be generally unordered, it is not unlikely for it keep the order of the original source, e. g. if the RDD was read from a file, the data will be partitioned while keeping the original ordering.

Compared to the standard MapReduce flow, Sparks does not execute the stages in batch, but whenever possible chaines them and simply feeds the result of a transformation to the next step. By doing this it avoids expensive persisting of the results, unless the programmer explicitly asks for it. It also makes sharing state across executors easier by enabling broadcast variables. These are values that are broadcasted to each executor. All of these features make Spark more useful for iterative algorithms than MapReduce.

The degree of parallelism of each job is given by a number of *partitions* of an RDD it is executed on. Each partition therefore contains a slice of the data contained in the RDD. To achieve the most out of the parallel execution and to evenly distribute the workload, it is recommended to have 2-4 times more partitions than available executors.

2.2.4 MLlib

MLlib is Sparks' collection of Machine Learnign algorithms as well as some helper algorithms used for linear algebra [14]. It ships with algorithms used for classification, regression, clustering as well as feature reduction and selection. Unfortunately, the library does not implement any semi-supervised algorithms.

The library is split into two main namespaces:

mllib This is a collection of lower-level APIs that accept and output RDDs. MLlib is the older part of the whole library and still widely used and actively developed. Apart from the machine learning algorithms it also contains the `linalg` package, which implements various datatypes and algorithms used for linear algebra.

ml This namespace groups together the new API. It is recommended to use the algorithms from this package [14]. They accept DataFrames as input.

The linear algebra library is particularly useful for us, as it does implement several distributed versions of data structures and algorithms. For our application we particularly need the implementations of distributed matrix data types and operations on them.

RowMatrix The simplest Matrix structure. It is row-oriented, however the rows do not have any meaningful indices. Each row represents a local vector, which is a limitation in case we have lots of columns.

Despite the fact that the rows are not indexed, it does support operations where it plays no role such as QR decomposition, SVD decomposition or column-wise operations such as computing column statistics.

IndexedRowMatrix A very similar structure to the RowMatrix but with indexed row. It supports a similar feature range as the RowMatrix. It is especially useful when one uses e.g. *BlockMatrix* but needs to perform row-wise operations, since it is possible to convert to/from other types of matrices.

CoordinateMatrix A matrix where each non-zero element is explicitly defined. It is particularly useful when both dimensions are huge and it is sparse. It is also quite useful as a temporary matrix when we know what it should contain. After it is constructed we convert it to `BlockMatrix`.

BlockMatrix This matrix is backed by an RDD of local matrices which represent a small block of the whole matrix. It is the most feature-complete type, as it supports operations between distributed matrices as opposed to the rest of the types, which do not.

2.2.5 Inspection tools

Spark offers a Web-based UI through which one can monitor and inspect the progress of a running job. The UI shows the stage in which the application is as well as various metrics about the stages and executors. It is also possible to visualise the execution as a directed acyclic graph. This graph represents the flow of the transformations and actions.

We have chosen Spark as our distributed engine. The key reasons, as described above, were:

1. A simple, yet powerful computational model. It keeps in line with the simplicity of MapReduce, but allows more freedom.

2. Rich APIs. As machine learning was one of the areas for which Spark was designed, it offers many API functions that are well suited for writing machine-learning algorithms.

Chapter 3

Implementation

The main parts of our implementations are these modules:

Data preprocessing This module is responsible for converting the source VOT files from LAMOST and CSV file with labels to a common format used in the subsequent stages. It also performs preprocessing of the spectra.

Graph models This module provides a standalone application as well as a reusable library for using Label Propagation and Label Spreading models.

3.1 Data preprocessing module

The data preprocessing module was written in Python using Python bindings in Spark. We have also made use of popular Python libraries NumPy, SciPy, Pandas and Astropy [15, 16, 17, 18, 19]. We have decided to use these libraries for their maturity, optimised operations and general acceptance in the Python community.

The initial configuration is driven by a JSON configuration file, whose description and example can be found in Section C.1. The module has two major functions:

1. Data conversion and integration
2. Data preprocessing

The data conversion has been tailored for the particular application for semi- supervised learning. The main input are the unlabelled spectra. They can be supplied as either a FITS binary table or a VOTABLE file. In our case, this would be the spectra from the LAMOST observatory. They are loaded using Sparks' `wholeTextFiles` function, which takes a path to a directory as its' intput and returns its' content as an RDD. Each spectrum is then loaded into Pandas DataFrame.

The module supports transformation of raw spectra to normalized spectra. Spectra are usually stored in logarithmic scale, so in order to get the physical representation, we need to transform the values to the linear scale.

The import phase also introduces the labelled spectra into the dataset. These can be either taken directly from the dataset, provided that we do the labelling either manually or crossmatch with some labelled spectra. Another option would be to directly import spectra from a labelled archive. We have chosen this course of action, and took labelled spectra from Ondejov's CCD-700 archive. Since there has been some work done with these spectra [20, 21], the VO-CLOUD already supports their conversion. Originally, the data are spectra stored in FITS files, sorted into directories by their type and star. The output of that module is a CSV file, where each line represents a spectrum. Each line starts by a unique spectrum ID, followed by a sequence of entries, where each entry is a given intensity on a particular wavelength. The line ends with a integer denoting a class, which can be between 0 and 3, included. It also produces a VOTABLE file, which contains metadata, such as the wavelength range. We take these final results as an input of the conversion process.

Since the spectra are from different sources, they were taken with a vastly different spectrographs with different resolution power. This means, that even though the spectra have been cut to the same wavelength range, the step between the measured points is different, resulting in a different length of the feature vector. To solve this problem, we have to resample the spectra with the higher resolution.

To resample the spectra, we first run them through a Gaussian convolution. The size of the kernel is chosen as a ratio of the resolution power of the two sources. The convolution is done to remove the fine details of the spectrum that the lower resolution spectrograph could not possibly capture. Then we interpolate the intensities of the higher-resolution spectra with the wavelengths of the lower-resolution spectra. The interpolation wavelengths are computed by taking the maximum of the lowest wavelengths and a minimum of the highest wavelengths. The step between them is taken as a mean step of wavelengths of all the spectra from the lower-resolution set. For the pseudocode, see Figure 3.

When importing new spectra, we also need to make sure that they are aligned with respect to the measured wavelength. There are inherent inconsistencies produced during the spectrography, so even though the spectra are *roughly* aligned, there are bound to be some errors. This can be seen when the spectra are plotted over each other. This would also affect the training of models, since properties we want to wach for, such as peaks and absorption would fall under different features. Fortunately, it is quite easy to remedy this. We can reuse most of the work done in the resampling procedure, albeit without the convolution. By interpolating over a common wavelength series,

Algorithm 3: Spectra resampling

```
Data: $L_o$: original low resolution spectra, $H$: high resolution spectra
Result: $L_h$: high resolution spectra that were resampled to the same
        resolution as $L_o$
begin
    $low, high$ = in parallel, aggregate over leftmost and rightmost values
    of wavelenghts of $L_o \cup H$ and return their supremum, resp. infimum
    $mean\_step$ = in parallel, compute mean(map($L_o$, $x$:
    $x.wavelengths - x.wavelengths$))
    $kernel\_size = resolution(H)/resolution(L_o)$
    $interp\_wavelengths$ = range from low to high with mean_step
    $interpolated$ = Map $H$ as $s$
        convoluted = Gaussian_convolution(s.intensities, kernel_size)
        interpolated =
        interp(s.wavelength, s.intensities, interp_wavelengths)
        yield interpolated
    return $L_h$
```

we align each of the spectra to one common grid.

The module also supports feature reduction by running a Principal Component Analysis on the data. PCA is widely used for feature reduction in many applications of machine learning. PCA transforms the original features into a set of linearly uncorrellated variables. The first component has the highest possible variance and any subsequent component has highest possible variance while also being orthogonal to the preceding components. We perform PCA using Sparks' parallel implementation. It is performed after the spectra have been resampled to the same resolution and aligned on the same wavelengths.

The result is then saved to a location specified by the user. The output location can be either on a local filesystem or on a distributed one, such as HDFS. The output is not a single file but a sequence of files, with each file per partition. Since we are assuming any other work done on the data shall be done in Spark, we are not concatenating the files, as Spark can work with such directory structures natively, modelling them as a single file.

3.2 Graph models

This module implements the graph models described in Section 1.4.1. The algorithms we have implemented are Label Propagation and Label Spreading. The pseudocode and general idea for these algorithms was already described in the above-mentioned section. Here we shall describe the details specific for implementing them on Spark and with regards to our application.

The application is structured as described here. The main function and initial data loading takes place in `GraphSSL` class. It parses the configuration JSON, loads the input data and applies an appropriate model on them, according to the passed configuration. The application expects a CSV file as an input. It should have the following structure: the first column must be a unique identifier. The preprocessing module uses the name of the spectrum. The values that follow are the intensities on a given wavelength. Note, that the actual value of the wavelength is of no interest for this algorithm. The last column should be a label signifying the type of a spectrum. Unlabelled spectra have a special value, in our case it is -1. All unlabelled spectra should be at the end of the file, because we rely on that when we want to keep the label distributions of labelled spectra unchanged.

The common parts of the algorithms, namely the construction of the distance matrix, transformations of input and output and others, reside in common class `GraphClassifier`. The interface of the class implements the interface of other algorithms contained in Sparks' `MLlib` library, so that it is possible to use it in pipelines relying on `MLlib`. That means, it implements method `transform`, which takes a `DataFrame` dataset as its' input and outputs a `GraphClassifierModel`. This model can be used for getting the transduced labels as well as for classifying new points.

Even though both algorithms support various graph kernels and distance functions, the sheer amount of data requires that we build a sparse distance matrix. If we constructed a dense graph, where each node would be a neighbour of other node, we would run out of memory soon enough, as the space complexity would be $\Theta(n^2)$ where n is the size of the dataset. Therefore we have decided not to implement Gaussian kernel or any other similar method and we build the graph using the KNN model.

The KNN itself can't be implemented using a naive exact algorithm, since that one would still need to compute distance between each data point, which would not be feasible. Instead, we use an implementation that build approximate KNN model using metric and spill trees [22].

3.2.0.1 Metric and spill trees

Metric trees are a variant of kd-trees. They model a spatial structure of points. Each node represents a set of points and has two children, with each representing a disjunct subset of its' parent. The root represents all points. A leaf node contains a small subset of points, potentially only one. The ideal way of partitioning a node is to find the two points, a v_l and v_r that are furthest appart, i. e. have the greatest pair-wise distance. However this has $O(n^2)$ complexity w. r. t. the size of the set. Therefore, heuristical approaches can be used. We might, for example, choose a random v_l and then find a v_r that

```
Function BuildMetricTree(points, leaf_size)
begin
    if size(points) ¡ leaf_size then
        return Node(points=points, left=nil, right=nil)
    Find $v_l$ and $v_r$ in points either using a heuristic, or finding an
    actual pair such that their distance is the furthest possible
    $r = v_l - v_r$
    foreach point in points do
        project point onto $r$
        add to projected
    median = median in projected
    foreach point in projected do
        if point is to the left of median then
            add point to left_subset
        else
            add point to right_subset
    return Node(points=points, left=BuildMetricTree(left_subset,
    leaf_size), right=BuildMetricTree(right_subset, leaf_size))
```

is furthest from it. We might switch between these approaches based on the size of the input.

After these pivot points are chosen we find a median point, v_m. This can be done by projecting all the points to $r = u_l - u_r$ and then choosing the median point from these. Now all the points that are to the left of this median point will go to the left child and the others will go to the right child. For the pseudocode of such a building procedure, see Function BuildMetricTree.

Searching on such a tree is done in an informed depth-first search. At each internal node, we choose left or right point first based on whether the point we are querying with is on the left or right side of the pivot. When used with KNN, we also maintain a list of k nearest points we have have encountered. At all levels it also checks whether any of the points can be nearer than already found nearest neighbours. If not, then it can safely prune this node from the search along with all its' descendants. The problem is, that the search can find a good-enough candidates quite quickly and then spend a lot of time searching through the tree and cutting the branches.

A heuristical approach to this search, called a *defeatist search* is that we do not attempt to find the very best candidates by searching through the whole tree, but instead we opt to find 'good-enough' points as our NNs. This can be done doing a simple BST search over the tree, therefore having $O(\log(n))$ time complexity. However, in the standard metric tree where the nodes have

Function BuildSpillTree(points, leaf_size, buffer)

```
begin
    if size(points) ¡ leaf_size then
        return Node(points=points, left=nil, right=nil)
    Find $v_l$ and $v_r$ in points either using a heuristic, or finding an
    actual pair such that their distance is the furthest possible
    $r = v_l - v_r$
    foreach point in points do
        project point onto $r$
        add to projected
    median = median in projected
    $l$ = plane parallel to median at distance $\frac{buffer}{2}$ to the left from it.
    $r$ = plane parallel to median at distance $\frac{buffer}{2}$ to the right from it.
    foreach point in projected do
        if point is to the left of $r$ then
            add point to left_subset
        else
            add point to right_subset
    return Node(points=points, left=BuildSpillTree(left_subset,
    leaf_size), right=BuildMetricTree(right_subset, leaf_size))
```

disjoint sets of points, this method can lead to highly incorrect results. This is evident when the query point q is close to the decision boundary, as then the probability that the NN of q is on the other side of the pivot point is almost the same, as the probability that it is on the side of q. Now a variant of the metric trees is called a spill tree. The spill trees are very similar to metric trees, except that they allow overlapping sets of points in sibling nodes. This may seem counterintuitive, as it increase the complexity of the tree, however it helps the heuristic search be more precise. The building of the tree is very similar to the metric tree, however in addition to the median point, we also choose points l and r such that they are both at distance d from the median point, either to the left, or right, respectively. Then, when partitioning, we assign each point to the left child if it is to the left of r and to the left child if it is to the right of l. Therefore we introduce a buffer zone of points, that are always searched, increasing the chance of finding a correct NN for a query point near to the decision boundary.

Spill trees thus offer a better chance of discovering good neighbours during the defeatist search. If our query point q is to the left of the decision point, we search the points right of l. Conversely, if it is to the right of the decision point, we search to the left of r. The points that belong to the buffer set are

Algorithm 4: Distributed metric tree building

```
Data: points: our dataset
begin
    Sample data and shuffle them to one machine
    Build a metric tree using the procedure BuildMetricTree.
    Map
        each element from points to a leaft in metric tree. Output the
        index of the tree as a key with the point
    Partition the keys such that they are evenly distributed across the
    computing nodes
    Shuffle
        Shuffle each point to a machine based on a key
    Reduce
        Run BuildSpillTree on each subset to create the leaf Spill Tree
```

searched in both cases.

The disadvantage of spill trees is that their depth might vary considerably, depending on d. With $d = 0$, the spill tree is a metric tree, as none of the points overlap. If, however $d > \frac{|v_l - v_r|}{2}$, then each child inherits the entire set of points of its parent and the tree growing might not even terminate.

A solution, which is used in our implementation, is called *hybrid spill trees.* It is a tree that combines both metric and spill trees. We set a threshold p, which determines, how we are going to partition a node. We choose a spill tree strategy by default, however if the fraction of shared points is greater, that a chosen p, we revert to a metric tree strategy. We label the nodes accordingly. Then, when searching through the tree, we decide on whether we use and exact DFS, or a defeatist search based on whether a node is a metric-tree node or a spill-tree node, respectively.

3.2.0.2 Parallel Metric and Spill Trees

To distribute this structure, we need to find a suitable partitioning [23]. While there is an option to simply partition the data into c parts, where c is a number of worker machines. This creates a need to search across all of the partitions during the search phase. This would be quite a waste of resources, for even when it is possible to search the partitions in parallel, there is a better way.

We can build a metric tree over a random sample of data, that fits into the memory of one machine. The leaf nodes of this metric tree are actually hybrid trees, built on data in their partition. The reason we force the top tree to be a metric tree is because it does not take as much memory space as a spill tree, since the points do not overlap. We also have to determine how big each of the spill trees should be. Ideally there should be at least one partition per computing machine, so there should be an upper limit u, such that $uc < M$, where M is a total memory available for the whole computing cluster. However, in practice, we may want to have more than one partition per machine to better distribute work.

Also note, that building the whole structre requires massive amount of data shuffling, as when it finishes, each partition must have the data physically present. This forces essentially each datapoint to move to the correct partition, causing massive usage of network, which may potentially slow down the process significantly.

The querying is then performed in a distributed manner. We find the correct spill trees using the top metric tree. Then we perform a search in parallel over the found spill trees, which will yield us the neighbours. To eliminate the need to backtrack on the metric tree, we do not search only on the assigned subtree, but also on the adjacent subtrees if the points there are within a certain threshold. Therefore, we can search multiple paths in parallel and only after they all return we choose the neighbours.

3.2.0.3 Graph construction

Running the KNN algorithm on our data yields a list of nearest neighbours. We then transform the list into a distance matrix $W \in R^{n,n}$ where n is the size of the data set. For each pair of neighbours i, j we compute their distance from each other and store it at $W_{i,j}$ and $W_{j,i}$. This yields a symmetrical sparse matrix, with at least k non-zero elements in each row. Note that the neighbourship relation by itself is not symmetrical, therefore we explicitly set both of the values at the same time. Spark does support several storage formats for matrices. `BlockMatrix` is the most feature-complete, supporting distributed matrix operations with matrices of the type, so we store everything by default in this format. However it is non-trivial to construct one, so we either use `IndexedRowMatrix`, which stores the matrix as an `RDD` of `IndexedRows` or a `CoordinateMatrix` which stores the matrix as an `RDD` of records element as a record of row position, column position and the value itself.

Everything was the same for Label Propagation and Label Spreading up unto this point. The difference between these two models, as pointed out in Section 1.4.1 is what graph representation we are using for propagating the labels and whether the original labels can change (LP) or cannot (LS).

3.2.0.4 Label Propagation

With Label Propagation we construct a transition matrix from the matrix created by the KNN algorithm, as described in Algorithm 5. A slight problem is that none of Sparks' distributed matrices support an inversion which we need for inverting the degree matrix D, which is then multiplied with the adjacency matrix. For performance reasons we chose not to do it using the inversion and

Algorithm 5: Construction of a transition matrix for Label Propagation

```
Data: dataset
begin
    Build a parallel hybrid tree for all the data
    Map each point in dataset to
        find nearest neighbours for point
        Output the index of a point and a sparse vector with its'
        non-null elements being the distances to the neighbours.
    transition_matrix=Map each row r to
        $sum = \sum_{i=1}^{n} r_i$
        transformed_row=foreach element in r do
            yield $\frac{element}{sum}$
        yield transformed_row
return transition_matrix.toBlockMatrix()
```

multiplication and instead we opted for a direct approach, where we compute the resulting matrix directly, as the only thing that needs to be done is dividing each element of the adjacency matrix in a row i with a corresponding element $D_{i,i}$.

Then we start iteratively propagating the labels in `labelPropagationRec`. The iterations stop when we either reach a user-defined number of iterations or when the label distributions converge. We check for the convergence of labels by computing the absolute difference between the newly computed label distribution and the one from the previous step, yielding a vector of differences between probabilities of each label. We then compute a mean difference, which we compare to a user-defined threshold. If the difference is at least this threshold, we consider the distributions to be converged and we finish the iterations. We also need to make sure to keep the original label distributions of labelled data unchanged, so in each iteration we filter out the original known labelling from the computed distributions and append them to the original labels. This iterative process returns the label distributions as a RDD of vectors. From these we can simply compute the inferred labels by returning the index of the vector element with the highest probability.

3.2.0.5 Label Spreading

This algorithm is very similar to Label Propagation, with the difference being that we use Graph Laplacian instead of a simple transitive matrix. The computation of Graph Laplacian is also a bit more complex, as it involves multiplication of three matrices, instead of just two. We again need to compute the inversion, but we use the same property of diagonal matrices as in Label Propagation. However, contrary to Label Propagation we do not need

Algorithm 6: Construction of the normalized Graph Laplacian for Label Spreading

```
Data: dataset
      $\alpha$
begin
    Build a parallel hybrid tree for all the data
    Map each point in dataset to
        find nearest neighbours for point
        Output the index of a point and a sparse vector with its'
        non-null elements being the distances to the neighbours.
    degree_rdd=Map each row r to
        yield $(r, \sum_{i=1}^{n} r_i)$
    graph_laplacian=Map each pair values $((r_1, d_1), (r_2, d_2))$ from
    degree_rdd to
        if $r_1$.index = $r_2$.index then
            yield MatrixEntry($r_1$.index, $r_2$.index,$\alpha$
        else if $r_1$.index $\neq$ $r_2$.index and they are adjacent then
            yield MatrixEntry($r_1$.index, $r_2$.index,$\frac{\alpha}{\sqrt{d_1 d_2}}$)
        else
            yield MatrixEntry($r_1$.index, $r_2$.index,0)
return graph_laplacian.toBlockMatrix()
```

to keep the original distributions, as these are controlled by parameter α as described in Section 1.4.1.2.

3.3 Problems and limitations we have encountered

There we several obstacles that we have encountered during the development. First, we were unable to setup a cluster in Ondřejov observatory, as no appropriate hardware was purchased. Thus, we had to rely on a Hadoop cluster in CESNET's Metacentrum.

First of them was linear algebra library in Spark. Internally it uses numerical JVM-compatible library *Breeze*[2], which offers powerful features and is inspired by Python's Numpy. Spark however wraps around Breezy's Vector and Matrix types without also exposing the same interface. This does pose some problems, as we needed to access some of Breezy's capabilities. For this we have created a collection of helper functions in `VectorUtils` and

[2] `https://github.com/scalanlp/breeze`

MatrixUtils. These convert Sparks' non-distributed vectors and matrices into Breeze's.

Another issue we have encountered is that the conversion from BlockMatrix to IndexedRowMatrix and vice-versa does not always yield the correct result. If a row in a given matrix is completely zeroed out, not even a reference to it is included in the destination matrix, leading to various problems when transforming the rows. Upon investigation, we have discovered that the ommision happens, because internally, the matrix is first converted to CoordinateMatrix and from this intermediary matrix it is converted to the desired format. Because of this we have implemented our own conversion in MatrixUtils, which includes even the rows which have been zeroed out.

3.3.1 Vocloud integration

Unfortunately we were not able to integrate with vo-cloud [3] because of insufficient hardware resources on their part. There is currently only one server that would be possible to use as a computing node and it makes no sense to setup a cluster there. New servers were supposed to be purchased and installed, but alas as of April 2016 they were not. Therefore we have decided not to integrate until the situation will change. The integration itself should not be, however, too difficult, provided that the cluster will be completely separate and will only serve for Hadoop/Spark jobs.

To integrate it, one would have to extend the Job class in vocloud project and write the appropriate logic. This would mainly consist of running the spark-submit command with appropriate parameters.

There would be some obstacles, however. As of now, Vocloud does not integrate with the Hadoop ecosystem at all, so we would need to either extend its' filesystem functionality to be able to seemlessly integrate with HDFS or implement at least the capability to send and receive data from HDFS filesystem. Vocloud now acts as sort of a resource manager, since it tracks which job runs on what worker. This would also have to change, since it would be highly unfair if this "spark-submitting" job would be considered equal to the other jobs, that actually do some work.

Chapter 4

Stellar spectra

In this chapter we shall describe the dataset, what exactly stellar spectra are and what challenges their processing poses.

4.1 The significance of a spectrum

Spectra are created by separating the light coming out of a star into different wavelengths. The intensity of the light on these wavelengths is then measured. The drops in the intensity are called *absorptions* and the raises *emissions*. The spectral line is also called a *continuum*.

Measuring a spectrum of a star reveals much about the nature of the measured object. Each element, that is present in the atmosphere of a star has its' own unique signature. This signature is represented as a series of absorptions or emissions at particular wavelengths. It may also reveal the stage of the evolution of that particular star.

4.2 Description of the dataset

The dataset is comprised of stellar spectra. The spectra are cut around the H_α line, which is about 656.28 nm. Our goal is to classify these spectra into four types based on the shape of the line.

- a pure emission (type 0)
- an emission with a central absorption (type 1)
- a pure absorption (type 2)
- an absorption line with an emission also called a shell line (type 3)

Representatives for the types are plotted in Figure 4.1. To illustrate the fine difference between type 2 and type 4 we provide detailed plot of the

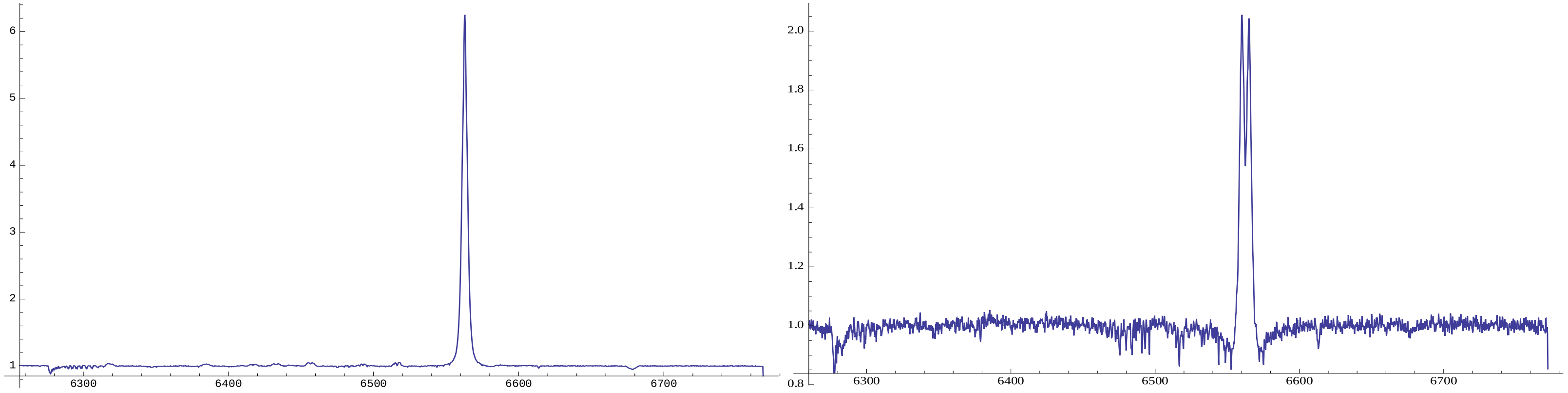


Figure 4.1: Examples of spectra with pure emission (left) and an emission with an absorption (right)

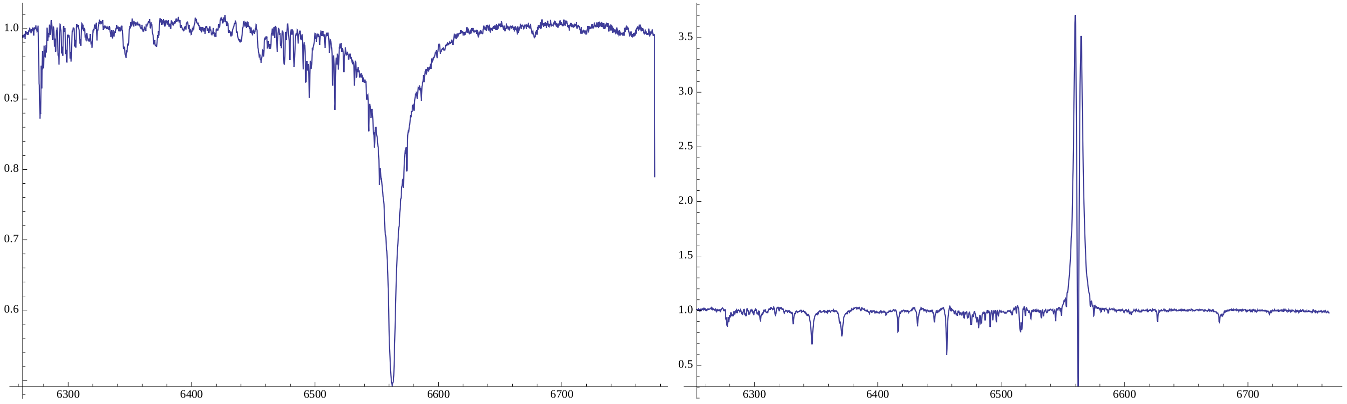


Figure 4.2: Examples of spectra containing a pure absorption (left) and an absorption line with an emission (right)

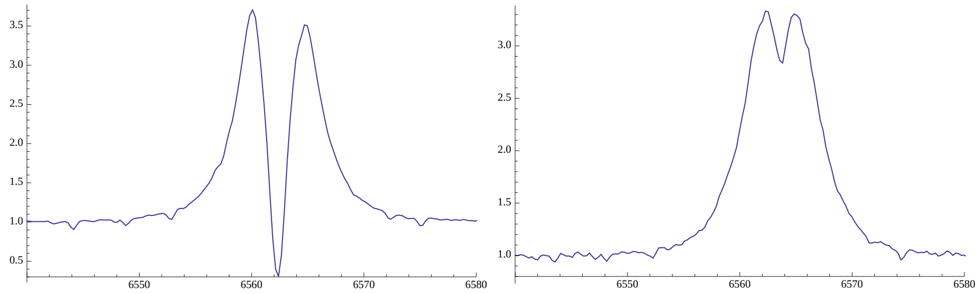


Figure 4.3: Details of spectra containing an absorption line with an emission (left) and an emission with a central absorption (right)

profiles for the an emission with a central absorption and an absorption line with an emission in Figure 4.3.

The dataset comes from two sources. The labelled part, having around 2000 samples, was taken in Ondřejov observatory using their 2 metre spectrograph. This spectrograph can measure one star at a time. We shall also refer to this spectrograph as CCD700. The spectral resolution power, is about 13000 in H_α. SRP, or R is given by equation $R = \frac{\lambda}{\Delta\lambda}$, where λ is a concrete wavelength and $\Delta\lambda$ is the smallest detectable difference between wavelengths. The resoluting spectra, cut around H_α have intensities measured on around 1800 wavelengths, from 6200 nm to 6800 nm. The spectra are quite detailed, with many small emissions and absorptions around the line, however the main distinguishing feature of different types, the central emission/absorption is very recognizable. The spectra on Figures 4.1 and 4.3 come from CCD700. The unlabelled data come from the first data release from LAMOST observatory in China. Their spectrograph is quite different from Ondřejov, as it can scan a huge area of sky at once, producing multiple spectra in parallel. The DR1 contains around 2 000 000 spectra, however we only work with around 1 000 000, due to technical reasons, specified in Section 5.1. The resolution power is smaller than the 2m spectrograph, so the spectra are fairly rough and not nearly as much detailed. We also lose the distinction between the pure emission spectra, and the double-peaked spectra. For this reason, we have decided to only have a binary label, with 0 meaning the spectrum is of an absorption type and 1 meaning it is of an emission type, which also includes the former types 1 and 3. For an example of how the resulting spectra look like, see Figure 4.4.

The labelled data were converted from FITS files to a CSV on the Vocloud system, using the preprocessing job that is already available there. Aside from the conversion they were also aligned to the same wavelength. This needs to be done as there are some inherent inaccuracies during the exposition of the spectra. If it had not been done, the features in each column would not actually have the same “meaning”. In addition, we also apply a Gaussian

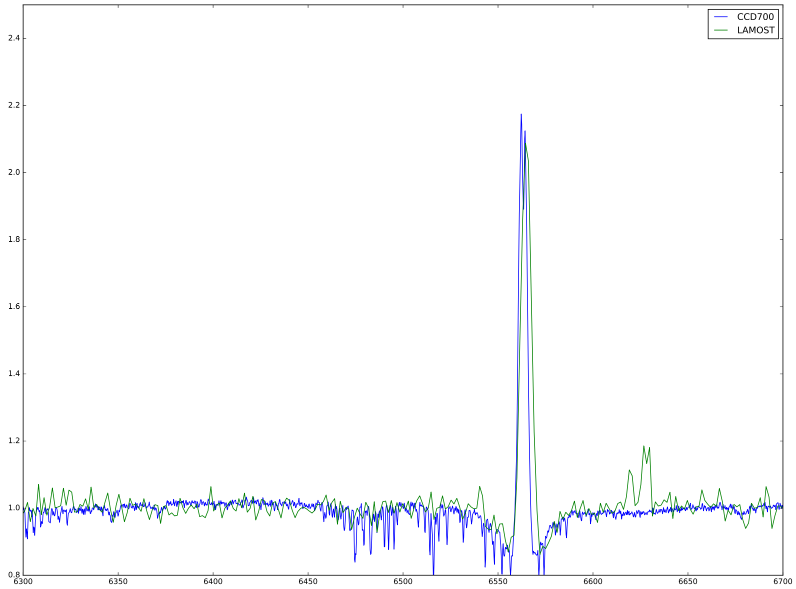


Figure 4.4: An example of a spectrum from LAMOST archive plotted against a spectrum of the same star from CCD700. Note that it depicts less finer details than CCD700 spectra.

convolution on the CCD700 spectra, so that they lose their finer details.

The result can be seen on Figure 4.5, where we plotted the same spectra as in Figure 4.4, however CCD700 spectrum has been resampled. Note that we have lost the double peak even in the original spectrum. This further supports our decision to merge the emission spectra into one single type.

The unlabelled data from LAMOST are stored as raw, logarithmic scale spectra in FITS format. The FITS files contain the entire measured wavelength, unlike CCD700, which only contained spectras cut around H_α. Both LAMOST and CCD700 spectra are fed to the distributed preprocessing job in Spark. It parses the FITS files, converts the spectrum from a logarithmic scale to the the wavelength scale. The user can also specify which part of wavelength they want to include. For our experiments, we have chosen the range of 6200 nm to 6800 nm.

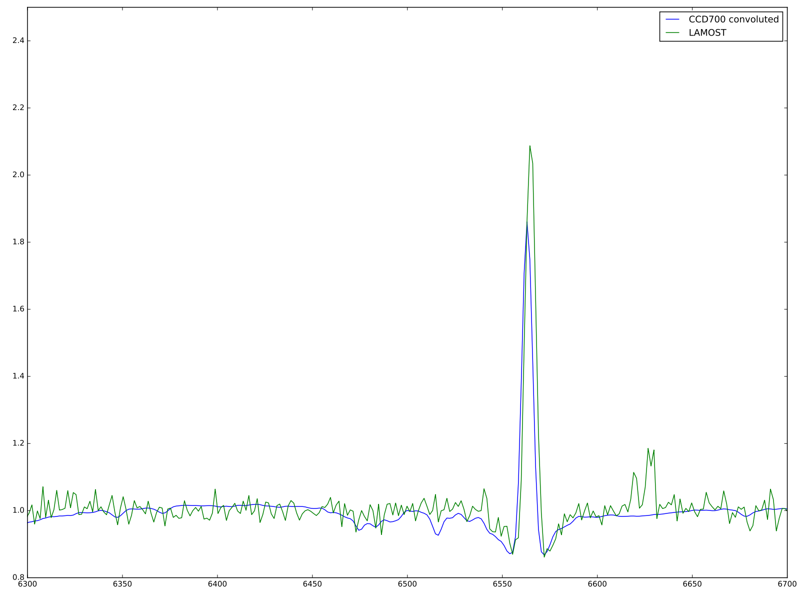


Figure 4.5: Spectra of the same star plotted against each other. The spectrum from CCD700 has been resampled.

Chapter 5

Experiments

In this chapter we describe the execution environment we used to run our experiments on as well as the methodology we used. We also present the results of our experiments.

5.1 Execution environment

Originally we were supposed to use machines installed in Ondřejov. However, they were not provided in time, so we have decided to find an alternative. We ended up deciding between using AWS service Elastic MapReduce (AWS EMR) or using the Czech academic grid Metacentrum.

AWS EMR is a managed service, where the user can provision a Hadoop cluster for themselves. A user can choose the type of the machines from the list of standard AWS instances, as well as the size of the cluster. The configuration of the cluster is done automatically, although user can customise parts of it. Even though this provides a very flexible way of provisioning a Hadoop cluster, we have decided against it for financial reasons.

Thus we have decided to use the Metacentrum grid. This grid is open for use by the whole Czech academic society, including researchers, students or teachers. We have chosen this They do have a operable Hadoop cluster that is open for all users of Metacentrum. The cluster has Hadoop 2.6 installed along with Spark 1.5.0. It is made of 24 computing nodes, each with 12 cores and 128 GB RAM. The HDFS storage has capacity of 1 PB, however the replication factor is 4 therefore the effective capacity is 250 TB.

Unfortunately we have also encountered several problems. First, the executor nodes do not have any common mountpoint and therefore the only shared storage is the HDFS. This means we also had to put the raw spectra on HDFS for them to be processed by the preprocessing job. This is far from an ideal use-case, as HDFS does not cope well with lots of small files for reasons described in Section 2.1.1. The fact that HDFS is the only common filesystem also posed problem with running the preprocessing job, as only some of the Python libraries were originally present on the executor nodes and having a way of storing all the common dependencies on a mounted filesystem would help us with that. There is also a limit to how many files can be in one folder, which is 2^20 in case of Metacentrum. Therefore, instead of the original 2 000 000 we only decided to classify a sample that would fit onto the filesystem.

5.2 Data survey

Before we begin with the experiments themselves, let's explore the structure of the data a bit to get a sense of how they look like. We do this to gain better understanding of the structure of the dataset as in how are the data distributed and how well the classess align to discrete clusters. We do this to test the Cluster Assumption mentioned in Section 1.3.2. We have used a small subset of the data with 10 000 samples.

To visualise the dataset, we used two methods to embed the data into a two dimensional space. The first method we used was the Principal Component Analysis transformation where we took the top two principal components. We have also used a method called TSNE, which was specifically designed to embed data in two- or three-dimensional space. It constructs a probability distribution over pairs of samples where the probabibility of being picked increases with the similarity of the objects. It constructs this distribution in both high-dimensional space and in the low-dimensional space. It then minimizes the *Kullback–Leibler divergence*, which is a measure of similarity between two distributions.

The results of the PCA decomposition are present in Figure 5.1. It is quite evident that the data do create two major clusters, with several spectra spread in between them. One of the clusters is composed of the members of the emission-type spectra, which includes also the double emission spectra and emission with absorption spectra. The second cluster is composed of absorption spectra and most of the spectra from LAMOST. This would suggest that most of the spectra in the LAMOST archive are of the absorption type. However there are some LAMOST spectra that belong to the emission cluster, which would suggest that they are the most likely candidates for being interesting spectra.

On Figure 5.2 we see a similar situation but perhaps with less clear clusters. This was created by a TSNE embedding. At the center is the cluster containing almost all the LAMOST spectra along with most of the absorption spectra. This structure is then surrounded by loose clusters of the other types and sometimes LAMOST spectra that are the most likely candidates for being of an emission type.

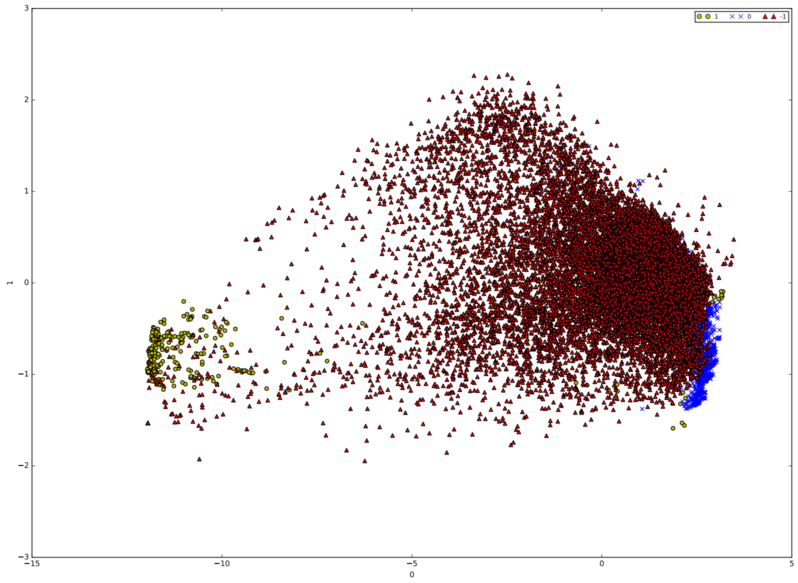


Figure 5.1: Structuring of the data after a PCA decomposition.

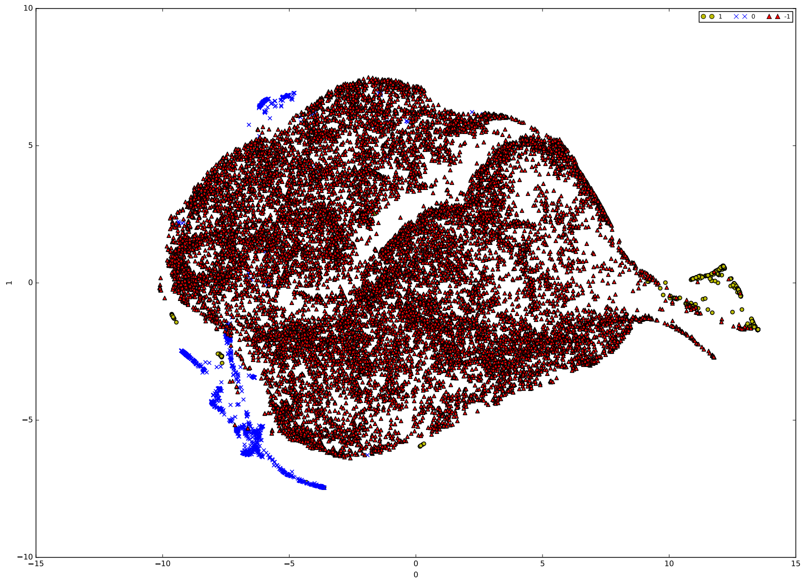


Figure 5.2: Structuring of the data after a TSNE embedding.

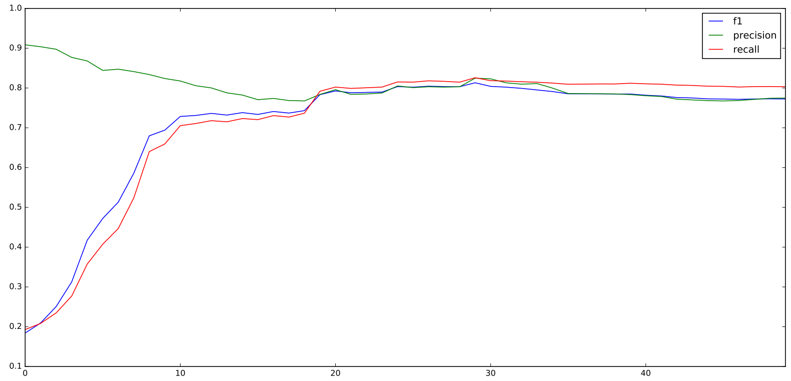


Figure 5.3: Dependency of the accuracy of Label Propagation with regards to the number of neighbours.

5.3 Experiments

We performed experiments on both the LAMOST dataset and on some of the well known datasets from UCI and similar repositories. We measured both the accuracy of the methods on a particular problem and also scalability in terms of both time and space requirements. We also compare with some other, supervised methods.

5.3.1 LAMOST DR1 dataset

Due to the sheer amount of data in the dataset we have chosen to measure the accuracy only on a small subset. We measure both the simple accuracy and the F1 measure. We also provide confusion matrix. To simulate the conditions of the usual semi-supervised learning problem, we have used a k-fold validation, where contrary to how it's usually done, we have used a larger testing set than a training set. The reason why we have used k-fold strategy is that it offers an easy way to test multiple subsets of the original data set and compute an average score. We measure these metrics for both Label Propagation and Label Spreading, with various settings of k and α.

5.3.1.1 Label Propagation

First we depict the performance of Label Propagation with regards to k, which can be interpreted as the density of a graph. We have measured the F1 score and accuracy for number of neigbours $1 \leq k \leq 50$ to find how it influences the classification performance of the algorithm. As expected, it rises with the k until a certain threshold, where it just hits a plateau or even decreases

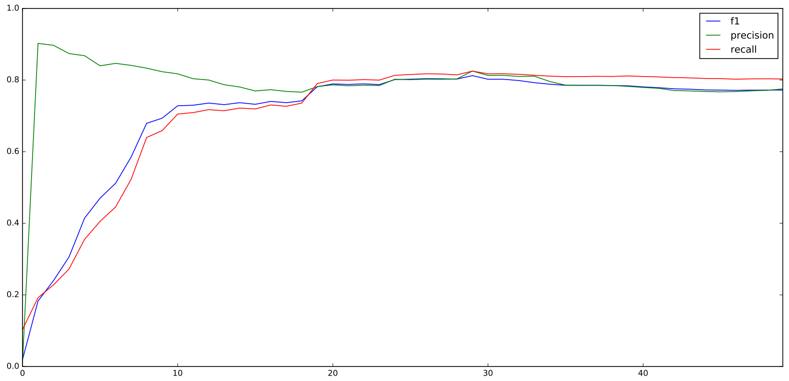


Figure 5.4: Dependency of the accuracy of Label Spreading with regards to the number of neighbours.

meaning that the model overfitted. We can also see, that the precision is fairly high for all values of k, meaning that *if* the model assigns a particular class, it has a high chance of being correct. However the recall is fairly low with low k since the model mis- identifies a lot of spectra. The results are presented in Table 5.1 and Figure 5.3.

Table 5.1: F1 score of Label Propagation for different number of neighbours. Bold k is the one with the best F1 score.

k	f1	precision	recall
0	0.184051	0.908555	0.192816
10	0.694096	0.823854	0.659328
20	0.783299	0.783908	0.791889
30	0.813179	0.824732	0.825840
40	0.784870	0.783486	0.811993
50	0.772476	0.774841	0.803129

5.3.1.2 Label Spreading

Next we measured the performance of Label Spreading with regards to the number of neighbours and setting of α. Recall, that α specifies how strongly the algorithm preserves the original labels in the training set. Setting of $\alpha = 1.0$ will force the original labels to stay the same. We have tested the number of neighbours in the same ranges as in the Label Propagation case. For α, we chose values $0.0 \leq \alpha \leq 1.0$ with step 0.1. We have tested the various

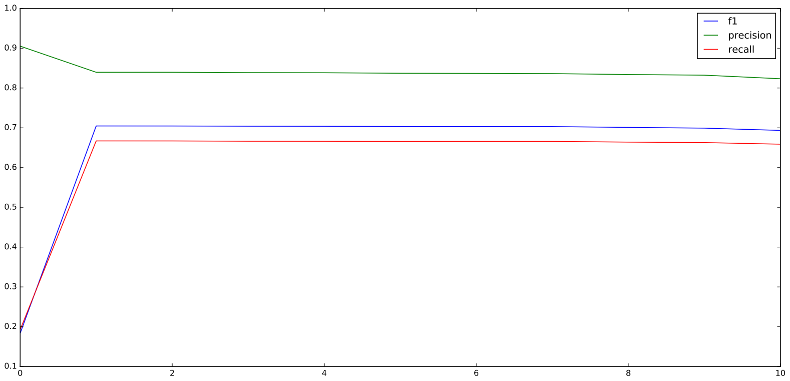


Figure 5.5: Dependency of the accuracy of Label Spreading on α.

settings of these values first separately and then with fixed best k and fixed best α.

As we can see on Figure 5.4 and Table 5.2, the progression of accuracy loosely follows the same path as in case of Label Propagation. We see that it slightly increases with the number of neighbours until it hits a threshold where the increase stops.

The results in Figure 5.5 and Table 5.3 show, that the performance of the model is more-or-less the same with regards to α, as long as $\alpha > 0.0$. This implies that spectra that are of the same type are actually quite close to each other and therefore reinforce the labels of each other quite a lot.

Let's move on to the measurements of computation time. We have taken this metric from metrics that the Sparks' web UI reports for each job that was run. We have measured the performance for various sizes of the dataset to measure how it scales in terms of number of data. We present results for both the preprocessing job as well as for both algorithms.

Table 5.2: Classificaiton performance of Label Spreading for different number of neighbours. Bold k is the one with the best F1 score.

k	f1	precision	recall
0	0.019282	0.010636	0.103129
10	0.693463	0.823321	0.658806
20	0.781369	0.781788	0.790324
30	0.812377	0.825084	0.825029
40	0.783979	0.782363	0.811472
50	0.772143	0.775240	0.802955

Table 5.3: Classification performance of Label Spreading for different value of α.

α	f1	precision	recall
0.0	0.183413	0.905268	0.192468
0.1	0.704628	0.839553	0.667034
0.2	0.704547	0.839571	0.666976
0.3	0.704055	0.838588	0.666280
0.4	0.703886	0.838368	0.666165
0.5	0.703259	0.837099	0.665759
0.6	0.703050	0.836747	0.665875
0.7	0.703032	0.836182	0.665817
0.8	0.701011	0.833884	0.664021
0.9	0.699154	0.832271	0.662862
1.0	0.693463	0.823321	0.658806

5.3.1.3 Preprocessing

The preprocessing job is mainly dependent on the size of size of the data. If we are not doing PCA transformation, there are also no user-set parameters that could influence either time or space complexity. We do not consider the time it takes to load or store the data in HDFS.

Table 5.4: Time scalability of preprocessing

sizes	time (seconds)	time increase
0.2	1080.0	1.000000
0.4	2230.0	2.064815
0.6	3830.0	3.546296
0.8	4320.0	4.000000
1.0	5423.0	5.021296

Table 5.4 and Figure 5.6 show that the preprocessing job scales quite well with regards to the size of the data. This is because the job is quite straightforward with nonexistent dependencies between the data. There is minimal amount of shuffling, which is done only at the beginning when we distribute the data evenly among the workers and at the end when we sort them by the label.

5.3.1.4 Label Propagation

The time of computation of Label Propagation also increases linearly. The slowest stages by far are the matrix multiplication stages. They are heavy on

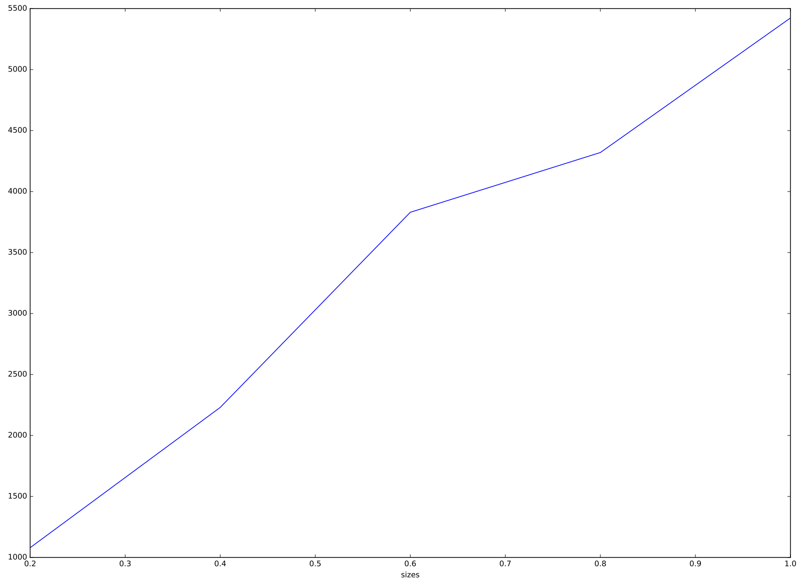


Figure 5.6: Time dependency on the size of the data for preprocessing.

Table 5.5: Time scalability of Label Propagation

sizes	time (seconds)	time increase
0.2	552.0	1.00
0.4	780.0	1.41
0.6	1020.0	1.85
0.8	4320.0	7.83
1.0	5423.0	9.82

the shuffling for they need to send the block matrices to all the appropriate machines in order to multiply them with all the correct positions.

5.3.1.5 Label Spreading

As we can see, the Label Spreading algorithm shows similar tendencies as Label Propagation. This is because at the core of the algorithms are very similar, and they simply multiply the matrices. However, the construction of the Graph Laplacian is quite a costly operation, as it needs to go over each possible pair in the distance matrix, therefore the job takes longer to finish.

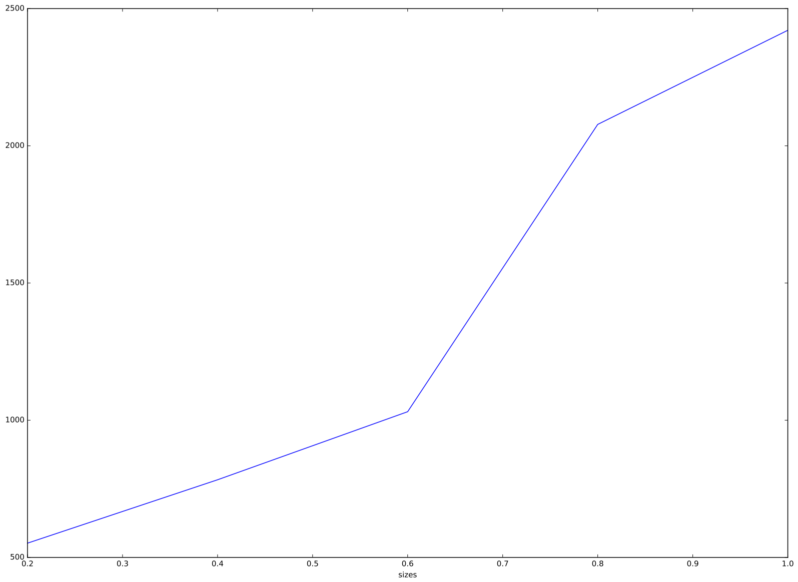


Figure 5.7: Time dependency on the size of the data for Label Propagation.

Table 5.6: Time scalability of Label Spreading

sizes	time (seconds)	time increase
0.2	2655	1.000000
0.4	10623	4.000126
0.6	25568	9.627965
0.8	49159	18.511108
1.0	86395	32.532321

5.4 Classification results

Before we present the results, let us reiterate the steps in the whole process once again:

1. We have converted the LAMOST spectra from FITS files. During this conversion, we have ran what is called a "rebinning to wavelength scale", since the spectra are originally stored in logarithmic scale.

2. We have performed a Gaussian convolution on the CCD700 spectra in order to remove the details that are not present in LAMOST spectra.

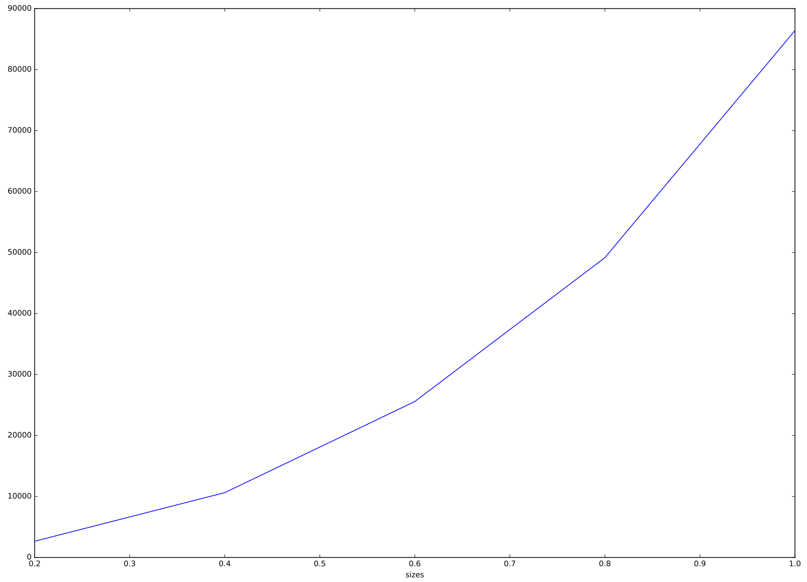


Figure 5.8: Time dependency on the size of the data for Label Spreading.

3. We have resampled all the data to one common wavelength axis. This is done in order to remove some inherent inaccuracies that can be introduced during when the spectra are taken.

4. We have cut the spectra around the H_α line.

5. We have applied a MinMax scaling on each spectrum individually. This caused the minimum value of a spectrum flux to be equal to 0 and the maximum value to be equal to 1.

6. We have binarised the labels, so that label 0 stands for absorption spectra and label 1 stands for all the other potentially interesting spectra.

7. Finally, the classification algorithm is run and we build our model that then classifies the candidates.

Recall, if you will, Figures 5.1 and 5.2. They show the structure of a small subsample of our dataset — some 10 000 spectra. Now let's review whether our graph algorithms can capture the structure.

As we can see on Figure 5.9, which corresponds to Figure 5.1 except with the training data removed, the classification does confirm what PCA visualisation suggested: that the data were in fact well separated into two major

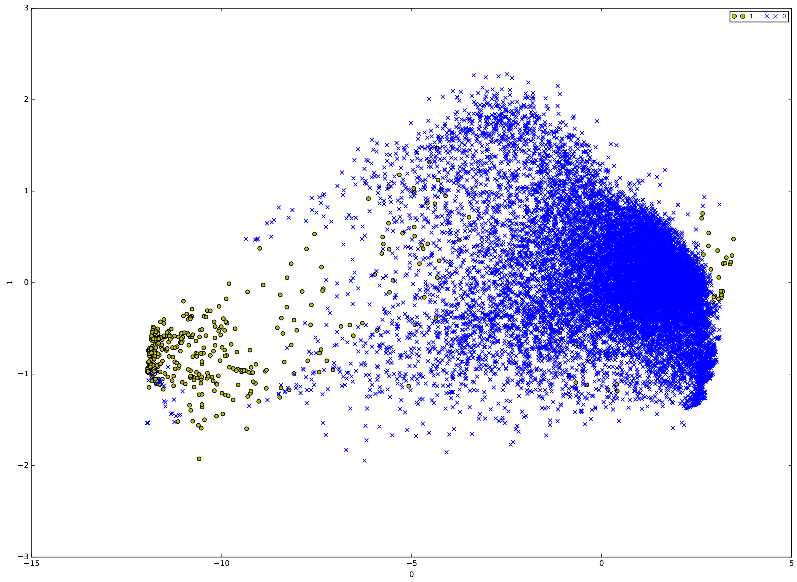


Figure 5.9: Classification visualised by PCA.

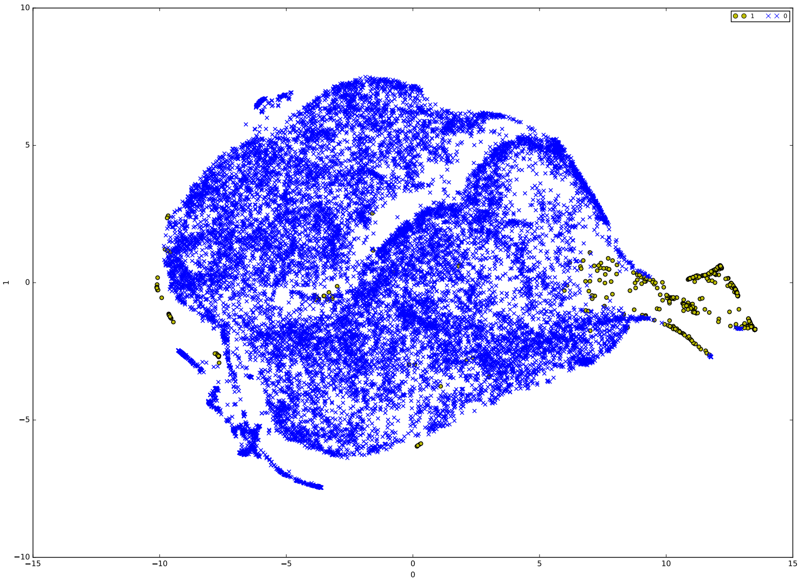
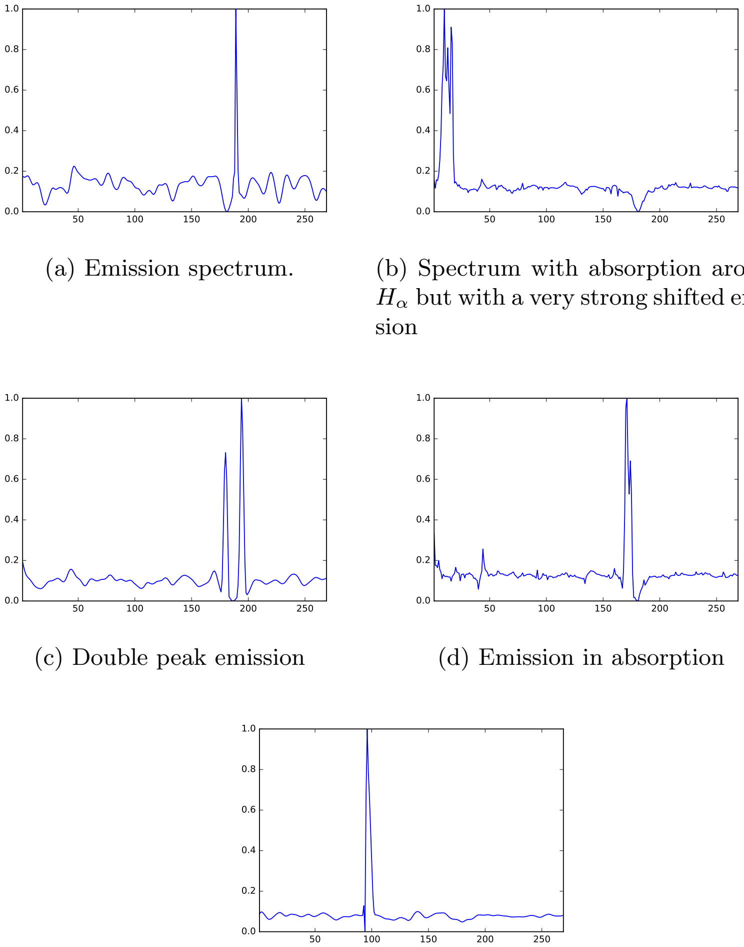


Figure 5.10: Classification visualised by TSNE.

clusters, with one containing the absorption spectra and the other containing the rest. It appears, however, that some spectra in both clusters were classified to a different type than the majority of spectra in said cluster. This would suggest that in reality they are more close to the spectra in the other cluster, but for some reason the PCA has transformed their features in such a way that they appear in the wrong cluster. It is also quite clear, that the amount of type 1 spectra is significantly less than of type 2.

The resulting candidate set did contain several interesting spectra. There are shapes, that were not present in the original labelled dataset, as seen on Figure 5.11. Appart from the expected emission spectra there were also shapes such as shifted emissions. The algorithm also classified several absorption spectra as emissions. These are the spectra that can be seen on Figures 5.9 and 5.10 as ones residing in the emission cluster.

This shows that the algorithm can really take advantage of the clustering assumption. It has correctly classified the spectra according to the clusters we see in Figures 5.9 and 5.10.



(a) Emission spectrum.

(b) Spectrum with absorption around H_α but with a very strong shifted emission

(c) Double peak emission

(d) Emission in absorption

(e) Shifted emission

Figure 5.11: Examples of detected spectra.

Conclusion

In this thesis we have attempted a semi-supervised learning-based approach to the problem of classifiying stellar spectra. The semi-supervised approach was chosen because algorithms belonging to this class were specifically designed to solve problems, where there are much more unlabelled data than labelled.

We have implemented a distributed preprocessing job, that converts the data from FITS files and optionally rescales them from logarithmic scale to linear wavelengths. It also applies a Gaussian convolution on high-resolution spectra from CCD700 so that they have similar level of details like the lower-resolution LAMOST spectra. This makes sure that we can actually use the CCD700 spectra as our labelled part of the training set. It also rebins them to one unified wavelength range.

We have implemented two graph-based classification algorithm called Label Propagation and Label Spreading. These algorithms create a graph representation of the data where the label propagates along the edges to the neighbours. Furthermore, we have implemented them on Spark so that we may cope with large amount of data.

The source codes for the preprocessing application and the graph algorithms are available on the attached electronic medium as well as on the following URLs:

Preprocessing `https://github.com/palicand/vocloud_spark_import`

Graph `https://github.com/palicand/graph_ssl`

We have ran the classification on approximately 1 000 000 of LAMOST spectra. Appart from the expected emission-line spectra we have discovered several unexpected and interesting spectra. The full classification result is stored on the attached electronic medium in form of a CVS file.

Bibliography

[1] Borne, K. D. Astroinformatics: data-oriented astronomy research and education. *Earth Science Informatics*, volume 3, no. 1, 2010: pp. 5–17, ISSN 1865-0481, doi:10.1007/s12145-010-0055-2. Available from: `http://dx.doi.org/10.1007/s12145-010-0055-2`

[2] Borne, K. D. Science User Scenarios for a Virtual Observatory Design Reference Mission: Science Requirements for Data Mining. *ArXiv Astrophysics e-prints*, Aug. 2000, `astro-ph/0008307`.

[3] Koza, J. *Design and implementation of a distributed platform for data mining of big astronomical spectra archives.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015.

[4] Chapelle, O.; Schölkopf, B.; Zien, A.; et al. Semi-supervised learning. 2006.

[5] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830.

[6] Seeger, M. A taxonomy for semi-supervised learning methods. Technical report, MIT Press, 2006.

[7] Kingma, D. P.; Mohamed, S.; Rezende, D. J.; et al. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems*, 2014, pp. 3581–3589.

[8] Dean, J.; Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, volume 51, no. 1, 2008: pp. 107–113.

[9] Senger, H.; Gil-Costa, V.; Arantes, L.; et al. BSP cost and scalability analysis for MapReduce operations. *Concurrency and Computation: Practice and Experience*, 2015: pp. n/a–n/a, ISSN 1532-0634,

doi:10.1002/cpe.3628, cpe.3628. Available from: http://dx.doi.org/10.1002/cpe.3628

[10] Ullman, J. D. Designing Good MapReduce Algorithms. *XRDS*, volume 19, no. 1, Sept. 2012: pp. 30–34, ISSN 1528-4972, doi:10.1145/2331042.2331053. Available from: http://doi.acm.org/10.1145/2331042.2331053

[11] HDFS Architecture Guide. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html, accessed: 2016-05-07.

[12] Apache Hadoop YARN. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html, accessed: 2016-05-07.

[13] Zaharia, M.; Chowdhury, M.; Franklin, M. J.; et al. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. Available from: http://dl.acm.org/citation.cfm?id=1863103.1863113

[14] Machine Learning Library (MLlib) Guide. http://spark.apache.org/docs/latest/mllib-guide.html, accessed: 2016-05-07.

[15] Walt, S. v. d.; Colbert, S. C.; Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, volume 13, no. 2, 2011.

[16] Jones, E.; Oliphant, T.; Peterson, P.; et al. SciPy: Open source scientific tools for Python. 2001–, [Online; accessed 2016-03-27]. Available from: http://www.scipy.org/

[17] McKinney, W. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, edited by S. van der Walt; J. Millman, 2010, pp. 51 – 56.

[18] McKinney, W. pandas: a Foundational Python Library for Data Analysis and Statistics.

[19] Astropy Collaboration; Robitaille, T. P.; Tollerud, E. J.; et al. Astropy: A community Python package for astronomy. *Astronomy and Astrophysics*, volume 558, Oct. 2013: A33, doi:10.1051/0004-6361/201322068, 1307.6212.

[20] Lopatovský, L. *Application of Self-Organizing Maps in Astroinformatics.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[21] Palička, A. *Application of Random Decision Forests in Astroinformatics.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2014.

[22] Liu, T.; Moore, A. W.; Yang, K.; et al. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, 2004, pp. 825–832.

[23] Liu, T.; Rosenberg, C.; Rowley, H. A. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *Applications of Computer Vision, 2007. WACV '07. IEEE Workshop on*, Feb 2007, ISSN 1550-5790, pp. 28–28, doi:10.1109/WACV.2007.18.

Appendix A

Acronyms

SSL Semi-supervised learning

SVM Support Vector Machine

TSVM Transductive Support Vector Machine

Appendix B

Contents of enclosed CD

```
candidates.zip ......... a zipped text file with the found candidates  .1
readme.txt ...................... the file with CD contents description
|
|
|__ src ..................................... the directory of source codes
|   |__ graph_ssl ..................... implementation of graph algorithms
|   |__ thesis ............. the directory of LaTeX source codes of the thesis
|   |__ vocloud_preprocessing ......... implementation of preprocessing job
|__ text ........................................ the thesis text directory
    |__ thesis.pdf ......................... the thesis text in PDF format
```

Appendix C

Configuration files

C.1 Data preprocessing

```
{
  ”input”: ”path/to/input”,
  ”output”: ”path/to/ouput”,
  ”labeled”: {
    ”file”: ”path/to/labeled/file”,
    ”metadata”: ”path/to/labeled/file”
  },
  ”pca”: {”k”: 40},
  ”partitions”: 40,
  ”label”: true,
  ”plot”: true,
  ”cut”: {
    ”low”: 6300,
    ”high”: 6700
  }
}
```

C.2 Graph job

```
{
  ”inputData”: ”path/to/files/generated/by/preprocessing”,
  ”outputData”: ”out.csv”,
  ”neighbourhoodKernel”: ”knn”,
  ”kernelParameters”: {”k”: 10, ”bufferSize”: 50.0}
  ”method”: ”LabelSpreading”,
  ”methodParameters”: {”alpha”: 0.9},
  ”partitions”: 24,
}
```