Integration of Probability and First Order Logic

THESIS

Submitted in partial fulfillment of the requirements of
BITS F421T, Thesis

by

Bhavul Gauri
(2011B4A7439G)

Under the supervision of
Dr. Ashwin Srinivasan
Deputy Director, BITS Pilani K K Birla Goa Campus

BITS PILANI - K K BIRLA GOA CAMPUS

December 7, 2015

Integration of Probability and First Order Logic

THESIS

Submitted in partial fulfillment of the requirements of
BITS F421T, Thesis

by

Bhavul Gauri
(2011B4A7439G)

Under the supervision of
Dr. Ashwin Srinivasan
Deputy Director, BITS Pilani K K Birla Goa Campus

BITS PILANI - K K BIRLA GOA CAMPUS

December 7, 2015

Acknowledgment

Besides requiring intelligence and hardwork, research requires equal creativity and patience. A pressurized mind or demotivated person would never be able to possess either. So, the first and the biggest thanks can go to no one else but my supervisor for this thesis, Dr. Ashwin Srinivasan. His continuous appreciation, support and guidance has kept me motivated all along; his patience and smile served me with flexibilities whenever I missed the deadline; his knowledge and his belief in me has been the back-bone of this research work. I gained a good amount of research related knowledge, motivation and insight about Artificial Intelligence, and it would not be so if he had not been there.

Further, I would like to thank the Head of Department of CS Department, BITS Pilani K K Birla Goa Campus for letting me have this opportunity of doing a Bachelor thesis, and for his continuous support throughout.

Last but not the least, My parents and my friends - those who have kept me sane on the days when I used to get frustrated because of getting stuck at a point in my work. I am highly thankful to them for being beside me all this while. This research work or for that matter, any of my achievements is not possible without them.

Certificate

This is to certify that the Thesis entitled, Integration of Probability and First Order Logic is submitted by Bhavul Gauri ID No. 2011B4A7439G in partial fulfillment of the requirements of **BITS F421T** Thesis embodies the work done by him under my supervision.

Signature of Supervisor

Dr. Ashwin Srinivasan
December 7, 2015 Deputy Director, BITS Pilani K K Birla Goa Campus

Abstract

Maximum Entropy modelling is a general purpose machine learning framework that has been tested to work well in various fields including spatial physics, computer vision and Natural language processing. However, Maximum Entropy model construction requires a set of features to start with. Real life datasets do not come in a proper constructed form of a set of features.

Inductive Logic Programming is a relatively new field that lies at the intersection of logic programming and inductive machine learning. Inductive Logic Programming can construct rules or hypotheses by learning from examples and background knowledge - something other fields of Machine learning can not do. All of this is done using an expressive first-order logic framework. The field has thus proven its worth in areas of inductive programming, knowledge discovery in databases, drug design and in Natural language processing.

One such Inductive logic Programming system is Aleph, written by Dr. Ashwin Srinivasan. Aleph has the capabilities of feature construction by learning from examples and background knowledge. This research work sources from the belief that Aleph can learn about the problem by examples and construct relevant features, which can then be fed to a Maximum Entropy model to correctly predict the random behaviour of the process. This study aims to combine the logical power of Inductive Logic Programming, and Probabilistic wonder of Maximum Entropy model. It serves as a step towards the big goal of combining logical learning and probabilistic learning, which is together called Statistical Relation learning, or sometimes Probabilistic Inductive Logic Programming.

Table of Contents

List of Figures

Chapter 1

Introduction

The field of Machine Learning has been busy and employed in working out methods to predict the behaviour of random processes, in finding rules, in pattern recognition, and in classification problems, etc. Various techniques have been coined for doing the same, some of which are Bayesian Networks, Neural Networks, SVM, Decision tree models, Maximum Entropy models. What all of these have in common is requiring as input the data in a form of a set of features. It has been observed that the better feature set you take initially, the higher is the performance with the machine learning task. This calls for a method to construct a good set of features from the data available, which is famously referred as feature construction.

1.1 Feature Construction

1.1.1 Features and Problems

In machine learning, a feature is an individual measurable property of a process under observation. Choosing a good set of informative features is crucial for forming effective learning models. Imagine judging how good a scientist is by their height and weight. Obviously, training such a model can not be expected to produce good accuracy in real life. Those are not the right set of features to predict the brilliance of a scientist. Real life problems, moreover, do not always present themselves in the form of a set of simple features; and if they sometimes do, the feature space is huge which makes constructing models complex and difficult.

For example, If a small text categorization problem was to be taken up where documents have to be classified into spam or not spam, feature space would intuitively consist of a set of unique words that appear in the documents. For a collection of even 10000 documents, hundreds of thousand of features can be expected. If pairs (bigrams) or words with their contexts are considered as features, then it may even result in millions of features. Building a model with such a huge set of features raises the complexity of the model.

With a huge set of features, not only does the complexity increase, but the performance of the model worsens. This is because not all the features of a problem are useful. Secondly, values of many features are correlated. These are called interrelated features. Whatever conclusive information can be extracted from one of these features is exactly what is extracted from the other, and hence, sustaining all of them in the feature space only adds unnecessary complexity to the model. So, finding a set of "good" features is necessary, and this is not always humanly possible, especially in case of huge feature space.

A bigger problem is most problems in real life do not output their results or data in the form of a table, or a set of features, comprising merely numeric, real or string values. Consider a problem of predicting DNA binding proteins. The knowledge of data would consist of the structure of proteins, and such knowledge does not exist on its own in the form of a fixed set of features. Inductive logic, however, can absorb this sort of background knowledge as well as swallow examples of real life to construct a set of features. We will be looking at that in detail in this thesis, as compared to other methods of feature construction.

1.1.2 Deductive and Inductive logic

Deductive logic assumes premises to be true and deals with determining what else would be true if the premises are true. In childhood when a kid is introduced to numbers and arithmetic operations, exercises include questions of divisibility. For example,

```
Q. Is 125 divisible by 5?
```

When such big numbers appear in front of a beginner, it becomes complex for him/her. At such times, one of the parents or the teacher goes on to tell:

```
"Any number that ends with 0 or 5 is divisible by 5."
```

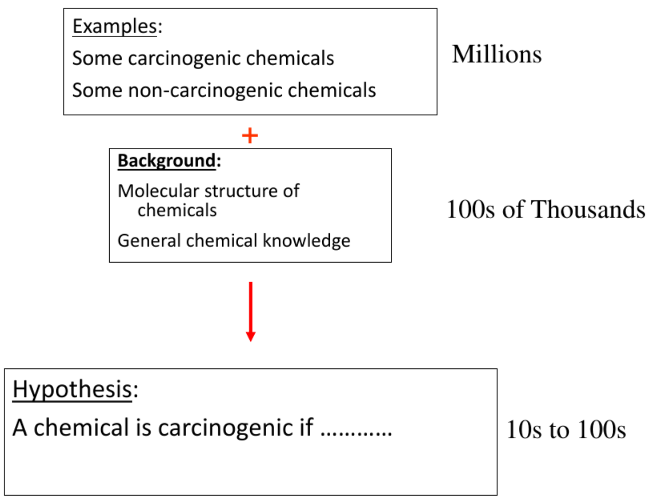
..and that gives rise to faster solutions due to deductive logic applied by the brain.

```
Any number that ends with 0 or 5 is divisible by 5,
and 125 ends with 5,
So 125 is divisible by 5.
```

It can be noticed that in deductive reasoning, we use a general rule to prove a specific statement. However, the Inductive logic goes from specific to general. Inductive logic starts with data and produces a general rule. Imagine an extremely early Mathematician studying numbers, writing them on the paper, and realizing that `1,2,3,4` were not divisible by 5, but `5` is; and then `6,7,8,9` are not divisible by 5, but `10` is, and so on...Maybe he went till 100, or till 1000, or till 10000 to study the pattern that

```
Every number that ended with a 0 or 5 only was divisible by 5.
```

So, here, a list of numbers was given, and it was known what digit they end with, and a general rule is induced from them. This is known as inductive logic. This general rule that is produced by inductive logic is often called a hypothesis.



Examples:
Some carcinogenic chemicals
Some non-carcinogenic chemicals
Millions
+
Background:
Molecular structure of chemicals
General chemical knowledge
100s of Thousands
Hypothesis:
A chemical is carcinogenic if ....
10s to 100s

Figure 1.1: Example of classifying a chemical as carcinogenic or not. This shows the real-world scenario of problems - how examples exist in huge numbers, background knowledge also exists in a good amount, however rules that can be the base of classification is in small numbers.[23]

Inductive logic programming(ILP) is a field that deals with programs that can induce hypothesis from examples and background knowledge, similar to how a Mathematician did above. Inductive inference lies at the core of Inductive logic programming. A glimpse of why this field is more so important can be seen above. While, inducing such a general rule of divisibility by 5 does not seem all so more important, imagine inducing rules about DNA proteins, about the suggestion for the next word as we are actively typing on our phones, about the carcinogenicity of chemicals. Inductive logic programming has already done all that by utilizing just background knowledge and examples. Another important point to observe is our world is filled with background knowledge and examples - we already knew of all natural numbers (examples), and we knew what digits they ended with (background knowledge). What lacks often is the knowledge of a hidden pattern, or a rule that satisfies them all. Figure 1.1 shows a glimpse of a real world problem, and displays why such hypotheses are important for us.

But how do these hypotheses help, one might ask. One of the main motives of machine learning algorithms has been of predicting; of solving classification, regression and pattern recognition problems. All of these predictions require some sort of deduction, for which we require a premise, or a rule, or a hypothesis to start with. Inductive logic programming can produce these premises.

1.1.3 Feature construction using ILP

Features are functions, and are described in detail in Section 4.3 of this report. Learning functions require at least first order learning. Inductive logic programming is one way to do first-order learning[24]. This makes ILP one way to construct features. Knowing that ILP can construct hypotheses, these hypotheses can be converted to relational boolean features quite easily. For example, imagine a problem of classifying essays as good or bad. Obtaining a seemingly large number of good and bad essays for training wouldn't be hard. Once that is done, ILP may find a set of hypotheses based on those.

One of them could be: `Good essays have 5 paragraphs.`
Another could be: `Good essays have no spelling mistakes.`

Once a set of good and informative hypotheses is output by ILP, these hypotheses can be converted to boolean features. For instance, the first hypothesis above can be converted to a relational feature f:

$$f(essaytext) = \begin{cases} 1 & \text{if the essay has 5 paragraphs} \\ 0 & \text{otherwise} \end{cases}$$

It is required however that only "good" rules/hypotheses are getting converted to boolean features and being added to feature space.

1.1.4 Other Methods of Feature Construction

Constructing features automatically without human intervention is a challenging task. Many of the attempts at it have been application specific. An ideal feature construction method would output a set of "good" features that:

1. improve accuracy of the model,
2. minimize the cost of computation,
3. are generalizable to different classifiers, and
4. allow for easy addition of domain knowledge

A number of different methods have been proposed, and they are briefed in this section. A detailed review and history of all proposed feature construction methods is presented in [25].

Decision Tree Related

Various decision tree related feature construction methods were developed. The first one being FRINGE, which started with an original set of features and then in each iteration combined existing features using boolean *negation* and *and* operators. It only however combined the features appearing at the fringe of the decision tree. CITRE and DC Fringe appeared shortly after FRINGE, and these used conjunctions and disjunctions to combine features taken from each positive branch.

The problem with these methods is of computational complexity. Since new features are added in every iteration, the number of features that need to be fed to decision tree algorithm become very large making the process computationally hard and inefficient.

Genetic Programming Related

Genetic Programming starts with a population of individuals, evaluates them based on some fitness function and constructs a new population by applying a set of mutation and crossover operators on high scoring individuals and eliminating the low scoring ones. In the case of feature construction, the initial population is a set of original features, fitness function is usually based on the performance of the classifier trained on these features. The genetic programming method essentially performs a search in new feature space to find a subset of "good" features. Many researchers have used this kind of approach for deriving a good subset of features for various applications, for instance, face recognition.

The only drawback of these evolutionary approaches is that some necessary or important features may get destructed due to the evolutionary process.

Annotation Based Approaches

Annotation based approaches require a domain expert besides the original set of features. A dynamic feature space is built by interactions between the learning machine and domain expert. During the training phase, learning machine presents some instance to the domain expert, the expert uses domain knowledge to provide additional annotations. The learner incorporates this additional knowledge and learns a model.

The obvious drawback here is a need for a domain expert through the whole process of feature construction.

1.2 Related Work

In this research work, maximum entropy modelling will be looked at, and features for this model will be constructed using Inductive logic programming. The inspiration for this work can been party credited to the early and promising work of Luc Dehaspe on MACCENT[26]. In his work, he modelled maximum entropy using clausal constraints. These clausal constraints are prolog queries evaluated on background knowledge written in Prolog. This was different than using usual features that come with the dataset. However, the set of these clausal constraints was assumed to be given in his modelling. This was still the first of its kind work which combined Inductive logic programming and maximum entropy. He also uses a feature selection algorithm based on approximation gain, to find an optimal subset of clausal constraints for best performance of maximum entropy model.

This sort of integration of features with probabilistic models has not died with time though. Even in recent research, computer scientists have tried combining features with probabilistic topic models. Topic models are a set of algorithms that aim to find hidden theme of collection of documents. The simplest kind of topic model is Latent Dirichlet Allocation (LDA) [27]. In 2007, Jun Fu Cai incorporated features constructed using topic models into modified Naive Bayes for word sense disambiguation[28].

Then in 2011, advantages of Probabilistic generative topic models were presented over discriminative models for multi-label document classification in [29]. In 2013, A. Srinivasan combined the capability of feature construction using Inductive logic programming with the powerful modelling of topic models for drug design problem[30].

The field that combines the expressive first-order logic for representation formalism and statistical machine learning models for modelling is called Statistical Relational Learning(SRL). Other names for it are Probabilistic Inductive logic programming or multi-relational data mining[31]. While for isolated problems like spam classification of emails or face recognition, machine learning models that exist are good enough, it is when problems get complex, SRL comes handy. For instance it might be more useful to know that an email was a non-spam mail and is a party invitation from a close friend rather than just knowing it was spam or not. In SRL, aim is not just to answer a classification problem, but to produce structured representations of data, involving objects described by attributes and participating in relationships, actions, and events.

Early SRL approaches were defined in terms of Bayesian networks, whereas recently most of the focus has been shifted to Markov logic networks. These networks generalize first-order logic and assign to all unsatisfiable statement a probability of 0, and to all the true statements (tautologies), a probability of 1. Each first-order formula or clause gets a weight between 0 to 1. Markov Logic Networks (MNL) have shown good performances in link prediction[32], entity resolution and information extraction.

This work shall restrict to a particular kind of SRL where features constructed using ILP are used to construct a Maximum Entropy model. The following chapters would talk of relevant fields, models and methods in required detail. For a more comprehensive review of research in the field of SRL or Probabilistic ILP, it is suggested to go through [33].

Chapter 2

Inductive Logic Programming

One area of computing that is not just improving the way we live every day but is able to fascinate normal people just as much as Einstein's theory of relativity does is Machine Learning. Practically, it is everywhere around us now - the smartphones we carry have built in software assistants, be it Siri on iPhones, Google Now on Android devices, or Cortana on Windows Phones, whose spine is full of Machine Learning algorithms that are learning from the way you use the internet and your device. Moreover, Machine Learning touches more aspects of our lives than we expect - Medicines, Robotics, Simulators, Web search, Face detection, Astronomy, Drug Discovery, Self-driving vehicles, Social Networks and much more. At the same time, it has been an equally booming field in academics in the last few decades with thousands of conferences and journals publishing developments in the field.

So, What is Machine learning, one might ask. Historically, Machine Learning was something of a reaction within Artificial Intelligence research. Even though companies use the terms Machine Learning (hereby referred to as ML) and Artificial Intelligence (hereby referred to as AI) very interchangeably, they are quite different. The Goal of AI is to create a machine that can mimic human intelligence, and while a big part of that is our 'learning capabilities', it also involves things like knowledge representation, reasoning, and even abstract thinking. Machine Learning is well-defined subset of AI that is the study of algorithms that can learn from experience. It is the science of making computers take relevant and correct actions themselves rather than explicitly programming them.

It has been quite a few decades of continuous innovation and development in the field of Machine Learning. There is a vast array of different machine learning techniques which today have become separate sub-fields by themselves - Decision Tree Learning, Neural Networks, Bayesian Learning, Support Vector Machines, and Inductive Logic Programming. Of all these, the one of relevance to this thesis is Inductive Logic Programming.

2.1 Introduction

Inductive Logic Programming is a Machine Learning approach that deals with developing tools and techniques to synthesize general rules from specific examples and background knowledge. Computers have been good at humongous numerical tasks like calculating and simulating how galaxies move; however, it becomes quite difficult for them to recognize the difference between a plastic bird and a real bird - something we learn to do quite easily and very well. Our brain is an amazing scientific machine that on noticing examples induces implicit rules or hypothesis - it does not tell you its equations, but keeps testing that hypothesis with newer and more examples and gets insanely good at it. Inductive Logic Programming is here to try to mimic that.

Inductive Logic Programming(from here on referred as ILP) is a combination of principles of inductive machine learning and representation of logic programming [1]. From inductive machine learning, ILP inherits its goal: to develop tools and techniques to induce hypotheses from observations (examples), and create new knowledge from experience [2]. By using computational logic as the representational mechanism for hypotheses and observations, ILP can overcome the two main limitations of classical machine learning techniques:

i. The use of a limited knowledge representation formalism (essentially propositional logic)

ii. Difficulties in using substantial background knowledge in the learning process

What this means is ILP enjoys a monopoly in the field of Machine Learning approaches by being the only one that uses an expressive First-Order framework instead of the simple attribute-value framework used by other approaches like Neural Network, Bayesian Networks, etc. Furthermore, ILP very conveniently also takes background knowledge into account. This makes it a wonderful blessing for things involving relational databases or dense background knowledge during learning, like drug design, or natural language processing, and time has proved this to be true.

2.2 Basic Example

Consider an example of learning about relationships in a family. Assume that it is known to us who is a parent of whom and the genders of people in the example. This will serve as our **background knowledge (B)**:

```
male(yash) ←
male(roy) ←
father(aditya, yash) ←
father(sunil, roy) ←
```

We are also given some **positive examples (E^+)**. Imagine positive examples as facts we have observed in real life. Hence these must be satisfied by our hypothesis/rule.

```
son(yash,aditya) ←
son(roy,sunil) ←
```

Similarly, we have some **negative examples (E^-)** too. These are facts known to us that can not happen in real scenario - they are known to be false.

```
son(roy,aditya) ←
son(yash,roy) ←
```

Believing **B**, and faced with examples (E^+) and (E^-), we might be able to guess very easily the following hypothesis $H_1 \in H$:

```
son(X,Y) ← male(X), parent(Y,X)
```

ILP Systems are able to guess this new piece of information we call hypothesis. This is the best hypothesis even we humans could form from the given dataset. How easy this seems to us just speaks of how great are our logical and learning capabilities. Since it was never told to computer what a son is and it is not so straightforward for a computer, the aim for it was to find a hypothesis that:

1. covers all the positive examples

2. does not cover any negative example, and

3. is sufficiently general.

Let's assume H to be set of hypotheses and contain an arbitrary number of individual speculations that fit the background knowledge and examples. Then, for as small a dataset that this is, hypothesis space is quite large.

A trivial hypothesis could be : `false`
This would cover no negative example, but also no positive example and hence would not be of any significant relevance.

Another trivial hypothesis could be : `son(X,Y)` $\leftarrow$
This would cover all positive examples but unfortunately all negative examples too, and that renders it useless.

Another hypothesis could be : `son(X,Y)` $\leftarrow$ $(X = yash \wedge Y = aditya) \vee (X = roy \wedge Y = sunil)$
This will cover all positive examples, and no negative examples, but fails for any new examples. This is considered too specific. It does not generalize.

As easy as it feels to us, synthesizing a hypothesis that is sufficiently general and correct with a probability that makes it easy to rationally accept it, is not an easy task. One observation that can be made is that background knowledge and hypothesis should together entail positive examples, and at the same time, not entail any negative examples. That would shrink the hypothesis space for us, but that is not all the conditions that need to be satisfied by a good hypothesis.

2.3 Theory

The problem in ILP is of inductive inference. Prior background knowledge B and examples E are given, where examples E consists of both positive examples and negative examples. The aim is then to devise a hypothesis H which satisfies several conditions related to the completeness and consistency with respect to the background knowledge and examples.

Figure 2.1 taken from [1] presents these conditions very clearly.

Consistency

It must be ensured that the problem has a solution. If one of the negative examples can be proved to be true from the background information alone, then the problem is not satisfiable. No hypothesis can nullify that effect. This gives rise to our first condition:

$B \cup E^- \not\models \Box$ **(prior satisfiability)**

Moreover, after adding hypothesis, it should still not be possible to prove a negative example. This gives rise to posterior satisfiability.

$B \cup H \cup E^- \not\models \Box$ **(posterior satisfiability)**

Completeness

The hypothesis that is induced should allow us to explain positive examples relative to the background knowledge. This means the hypothesis should fit the positive examples given. This gives rise to posterior sufficiency.

$B \cup H \models E^+$ **(posterior sufficiency)**

And of course, there would not be a need for a hypothesis if Background Knowledge alone can entail all the positive examples. This gives rise to Prior Necessity.

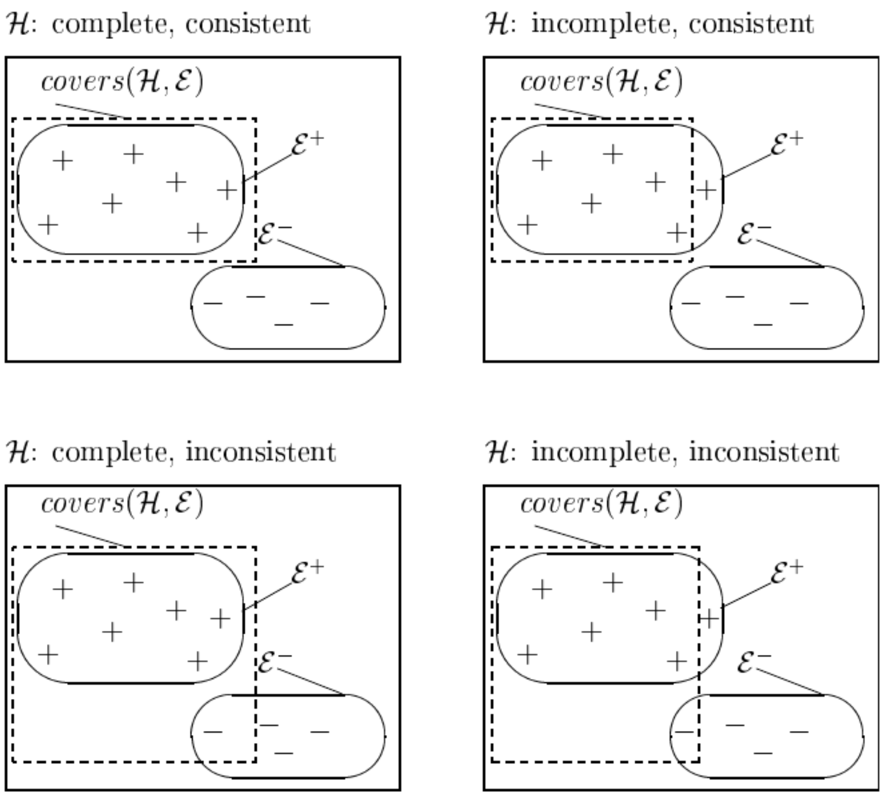

$\mathcal{H}$: complete, consistent
$\mathcal{H}$: incomplete, consistent
$\mathcal{H}$: complete, inconsistent
$\mathcal{H}$: incomplete, inconsistent
$covers(\mathcal{H}, \mathcal{E})$
$\mathcal{E}^+$
$\mathcal{E}^-$

Figure 2.1: Completeness and consistency of a hypothesis

$B \not\models E^+$ **(prior necessity)**

So, a good hypothesis must satisfy these four conditions. It should be noted that background knowledge, examples and hypotheses can be any well-formed first order formula.

2.3.1 Normal Semantics

So, in this general setting that we talked about, given B and examples $E = E^- \cup E^+$, aim is to find a hypothesis H that follows the following constraints: Prior Satisfiability, Posterior Satisfiability, Prior Necessity and Posterior Sufficiency. Such a general setting is called Normal Semantics.

2.3.2 Definite Semantics

A clause is nothing but a disjunction of literals. A Horn clause is a clause with at most one positive literal. A definite clause is a clause with exactly one positive literal. So, a definite clause might look something like:

$$\neg a \vee \neg b \vee ... \neg g \vee h$$

What this also means is each definite clause can be represented in a *this-if-that* form. For example, the above general definite clause could also be represented as:

$$h \leftarrow a \wedge b \wedge ... \wedge g$$

Even though Normal Semantics looks sound and correct, practically in most ILP systems, background knowledge and hypothesis is restricted to being a definite clause. As noted by Muggleton de Raedt (1994, pp. 635-636), this restriction simplifies the problem of ILP, since for every definite program T there is a model $M(T)$ (its minimal Herbrand model) in which all formulae are decidable and the

Closed World Assumption holds.

In this setting, a different set of conditions is required to hold. Since all formulae will have a truth value in the minimal model, every formula in E can be referred. In this setting, the conditions to be satisfied by hypothesis H are:

Consistence

- **Prior Satisfiability**: all e in E^- are false in $M^+(B)$.
  This ensures that negative examples do not become a part of the minimal model of background theory and the hypothesis.
- **Posterior Satisfiability**: all e in E^- are false in $M^+(B \cup H)$.
  This ensures that negative examples are not being satisfied by the hypothesis.

Completeness

- **Prior Necessity**: some e in E^+ are false in $M^+(B)$.
  This necessitates the existence of hypothesis. If this condition is false, the hypothesis would not add anything new or substantial information.
- **Posterior Sufficiency**: all e in E^+ are true in $M^+(B \cup H)$.
  This ensures that the hypothesis is able to explain all the positive examples in the minimal model.

An additional constraint that ILP systems impose in definite semantics is to allow only true and false ground facts as examples. This means causal examples can not be a part of positive or negative examples. This implies that something like `son(aditya,yash)` is acceptable. However, `father(X,Y)` or `father(yash,aditya) ← male(yash), son(aditya,yash)` is not allowed. When this condition is imposed, the setting is called **Example Setting** (as a subset of Definite Semantics).

2.3.3 Non-monotonic Semantics

In Non-monotonic Semantics, the background knowledge is a set of definite clauses and there are no examples. The positive examples are a part of background knowledge itself, which we denote by B', where $B' = B \cup E^+$. There are no negative examples either.
In this setting, the task of ILP is to find a hypothesis such that the following hold[3]:

- **Validity**: All clauses belonging to a hypothesis hold in the background knowledge B, i.e., they are true properties of the data
- **Completeness**: All information that is valid in the minimal model of B' should follow (can be entailed) from the hypothesis.

2.4 Classification of Different ILP Systems

Over the years, many ILP systems have been developed that work in very different ways to solve the same ILP problem. MIS[4],CIGOL[5], GOLEM[6], FOIL[7],CProgol[8], Indlog[9] and Aleph[10] to name a few. As a part of this study, we shall be reviewing Aleph only.

All of these can be classified based on the following concepts:

Single/Multi-predicate learner
In case of Single predicate learner, E contains examples of only the predicate for which hypothesis would be induced. On the other hand, in Multiple predicate learner or theory revision case, the aim is to learn a set of interrelated predicate definitions and hence examples E can contain different predicates.

Batch/Incremental learner
In the case of a Batch learning ILP system, all examples (E) are provided prior to the learning phase. In Incremental ILP systems, examples are given one-by-one and simultaneously training takes place.

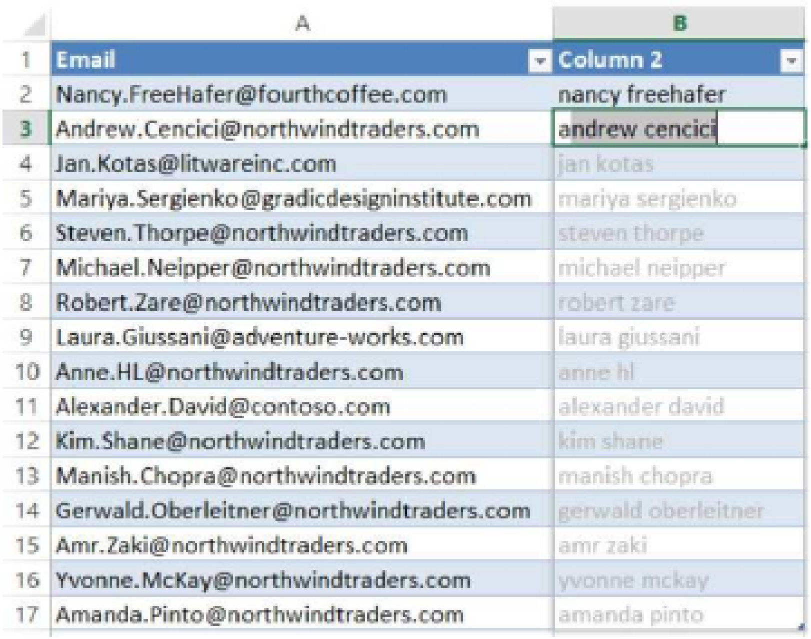
Interactive/non-interactive learner
In interactive systems, ILP system requires the user to verify the validity of the hypothesis it is inducing, or the examples it is generalizing. In non-interactive learner, this is not the case.

Further, ILP systems can also be divided on the base of whether they can make up new predicates or not, whether they accept an empty hypothesis or not, and whether they require to have background knowledge containing only ground facts and no variables or causal relations.

However, nowadays, the basis of classification has been reduced to just one dimension. ILP systems are divided into two - **Empirical ILP systems** and **Interactive ILP Systems**. Empirical systems are non-interactive systems and follow batch learning, and are known to learn one predicate from the pool of examples. Interactive systems, however, can learn multiple predicates from examples and by interacting with the user.

2.5 Applications and Future of ILP

Since the 1990s, ILP has been used to tackle real-life problems in various domains: in predicting secondary structure of proteins using GOLEM 11], in creating temporal diagnostic rules for satellite component failures [12], in drug-design [13], learning rules for finite element mesh design using mFOIL [1], and so on. Time and again, ILP proved its relevance due to the relational descriptions ILP generates.



	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

Figure 2.2: Flash Fill : An Excel 2013 feature that automates repetitive string transformations using examples. Once the user performs one instance of the desired transformation (row 2, col. B) and proceeds to transforming another instance (row 3, col. B), Flash Fill learns a program `Concatenate(ToLower(Substring(v,WordToken,1)), ,` `ToLower(SubString(v,WordToken,2)))` that extracts the first two words in input string v (col. A), converts them to lowercase, and concatenates them separated by a space character.

Recent developments such as Flash Fill feature show that ILP is not just made for complicated relational databases learning problems. In Microsoft Excel 2013 (or later version), there exists highly prominent Flash Fill feature, which automatically understands the pattern in the input and generates string processing programs internally so as to offer a suitable suggestion. (See figure 2.2). This is an example of Inductive programming. Inductive Programming studies the automatic generation of computer programs from positive and negative examples and background knowledge.

Over the last decade, the internet has exploded and so has the gadget market. At all times, the websites and apps that we use can monitor our usage and treat them as examples to learn rules from them. The website IFTTT (short for If this then that) has these *recipes* which basically represent rules in the form of actions. For example, say whenever one posts a picture on Instagram, it might be required to upload it to the Dropbox account as well. Such a recipe can be built by IFTTT and used. However, with the power of ILP, as smartphones usage data is sent to major tech giants, they can learn from the usage and built these sort of rules themselves. So that, next time when one posts a picture on Instagram, maybe phone itself will ask if it could be saved to Dropbox too. This leaves no bounds for ILP's applications to the internet.

It has been just over two decades and ILP has left a mark in drug design, in learning rules from chess databases, in fields like inductive programming, in mesh design. The ability of ILP to accommodate background knowledge gives it an edge, and if something can be said about ILP for future, it is that with proper research and utilization of the first-order framework that ILP uses, the possibilities are limitless.

Chapter 3

Aleph - an ILP System

3.1 Introduction

Aleph is short for A Learning Engine for Proposing Hypothesis [10]. It is an open source interactive ILP system written in Prolog by Ashwin Srinivasan with the main intention of exploring ideas. It builds on the idea of inverse entailment presented in [14].

Aleph is written in Prolog and can run with Yap Prolog Compiler as well as SWI-Prolog. It works on the principles of theory of ILP that were discussed in the previous chapter. Besides constructing first-clausal hypothesis, Aleph also lets the user choose the order of generation of the rules, change the evaluation function, and the search order. It also features incremental learning, and feature construction.

Recently, capabilities have been added to Aleph to emulate functionalities of other ILP systems too, like Progol, FOIL, FORS, Indlog, MIDOS, SRT, Tilde WARMR. This makes it suitable for carrying out ILP research experiments with different methods. In addition to all this, Aleph has been made free for research and teaching. All of this makes it a powerful resource for working in ILP.

3.2 Basic Aleph Algorithm

At Aleph's heart is a basic algorithm which can be described in 4 steps:

1. **Select Example:** Select an initial example to generalize. If there are no more of these, stop, otherwise proceed to the next step.
2. **Build most specific clause:** Construct the most specific clause that entails the example selected in the previous step, and is within language restrictions provided. This is usually a definite clause with many literals, and is called the bottom clause. This step is sometimes called the saturation step.
3. **Search:** Search for a more general clause than the bottom clause by using Branch and Bound algorithm. Subsets of literals in the bottom clause are observed and the one with the best score gets chosen. Two points should be noted. First, confining the search to subsets of the bottom clause does not produce all the clauses more general than it, but is good enough for this thumbnail sketch. Second, the exact nature of the score of a clause is not really important here. This step is sometimes called the reduction step.
4. **Remove Redundant:** The clause with the best score is considered as a hypothesis and is added to the current theory, and all examples made redundant are removed. This step is sometimes called the cover removal step. Note here that the best clause may make clauses other than the examples redundant. Again, this is ignored here. Return to Step 1.

Aleph also allows the user to modify parameters related to each of these steps for an advanced user. However, the most basic use of Aleph is for constructing theories. Aleph uses three files to construct a theory - A background knowledge file (uses extension `.b`), a positive examples file (uses extension `.f`) and a negative examples file (uses extension `.n`)

3.3 Aleph Terminologies

3.3.1 Background Knowledge File

Using the extension `.b`, a background knowledge file contains all the background knowledge to be loaded into Aleph. This includes prolog clauses, directives to consult other files, and language as well as search restrictions. These language and search restrictions are added through modes, types, and determinations.

3.3.2 Positive examples file

Positive examples of the data to be learned by Aleph are written in a file with a `.f` extension. The filesystem should be the same as that used for background knowledge. Positive examples are nothing but ground facts, as opposed to containing any variables. Here's an example of positive examples file:

```
brown(bear).
brown(cat).
green(caterpillar).
```

Aleph also contains the code for dealing with non-ground positive examples, but that has not yet been tested rigorously.

3.3.3 Negative examples file

Negative examples of the data to be learned by Aleph are written in a file with a `.n` extension. The filesystem should be the same as that used for background knowledge. A negative examples file is optional. Aleph is capable of learning from positive examples only.

Negative examples, just as positive examples are also ground facts. They are used to tell Aleph that whichever facts are asserted in the file are known to be false. An example of such a negative examples file would look like this:

```
brown(rabbit).
green(monkey).
green(human).
```

3.3.4 Mode declaration

The mode declarations in the background knowledge file (filename.b) describe the predicates that can appear in any clause constructed by Aleph. The declaration specifies the kind of arguments for each predicate and also informs Aleph if the predicate is to be used in the head (modeh declaration) or in the body (modeb declaration) of the generated clause. The general format for mode declaration is:

```
mode(RecallNumber, Mode)
```

The **RecallNumber** sets the limit number of alternative instances of one predicate. A predicate instance is a substitution of types for each variable or constant. This number can take any positive value greater than or equal to 1. If it is not known how many instances can be allowed, it is usually set as `*`. However, if the limit of possible solutions for a particular instance needs to be bounded, it's possible to define them by RecallNumber. For example, if we want to declare a predicate `parent_of(P,D)` the recall should be 2, because the daughter D, has a maximum of two parents P.

The **Mode** indicates the predicate format, i.e. its arguments and whether they are input or output or constant. Each Mode has the format:

```
p(ModeType, ModeType, ...)
```

Each ModeType is either simple or structured.

A simple ModeType is one of :

(a) **+T** meaning that whenever a literal with predicate symbol p appears in a hypothesized clause, the corresponding argument should be an input variable of type T.

(b) **-T** meaning that whenever a literal with predicate symbol p appears in a hypothesized clause, the corresponding argument should be an output variable of type T.

(c) **#T** meaning that it should be a constant of type T.

A structured ModeType is of the form `f(..)` where f is a function symbol, each argument of which is either a simple of structured ModeType.
Following are few examples of modes:

```
:- mode(1, mult(+integer,+integer,-integer)). % simple ModeType
:- modeh(1, uncle_of(+person,+person))
 % uncle_of should
appear in head.
:- modeb(*, sister_of(+person,-person))
 % sister_of should
appear in body, can have any number of alternate instances
:-mode(1, mem(+number,[+number|+list])). % structured modeType
```

3.3.5 Types

Types have to be specified for every argument of every predicate as a part of mode declarations. *Note that there is no pre-defined set of types that exists in Aleph.* For Aleph, types are just named, and no type-checking is done. Every type differs by name from each other and is treated distinctly, even if one is sub-type of another.

For example: `integer` could be a type and `decimal` could be a type too, which could be used as `integer(5)`, `decimal(2.2)`, `decimal(50.0)`, or `integer(X)`, etc.

3.3.6 Determinations

Determination statements declare the predicate that should be used to construct a hypothesis. They usually appear in the following format:

```
determination(TargetName/Arity,BackgroundName/Arity).
```

The first argument describes the name and arity of predicate that will appear in the head of the hypothesis, and the second argument is name and arity of a predicate that can appear in the body of the hypothesis. Usually, there will be many determination declarations for a target predicate since most of the times, more than one choices exist for the predicate in the body.
Note that if no determinations are present Aleph does not construct any hypothesis.

Further, determinations are only allowed for one target predicate on any given run of Aleph: if multiple target determinations occur, the first one is chosen.
Here are some examples:

```
:- determination(uncle_of/2, parent_of/2).
:- determination(uncle_of/2, brother_of/2).
```

The first one here means hypothesis to be constructed should have an `uncle_of` predicate as the head and tells that it takes 2 arguments, and that the body may have a `parent_of` predicate which also takes two arguments. The two determinations combined say that `uncle_of` predicate would be in the head of the constructed clause and the body may have one (or both) of `parent_of` or `brother_of` predicate.

3.4 Working with Aleph

3.4.1 Loading Aleph

Every version of Aleph has been contained in a single file, usually called alephX.pl where X stands for the version number. The current version, as of 2015, is the 5th version. You need to load aleph in your Prolog compiler. Aleph runs best with yap compiler, however, it has also been tested with SWI. It is suggested that you should load the compiler with sufficient stack and heap size.

For example, for loading it in Yap with a stack size of 5000 K bytes and heap size of 20000 K bytes:

```
username:$ yap  s5000  h20000
% Restoring file /usr/lib/Yap/startup.yss
YAP 6.2.2 (x86_64-linux): Sun Nov 24 20:27:47 UTC 2013
?- [aleph].
% consulting /mnt/F632E52432E4EB17/ACADEMICS/5-1/Thesis-
work/ALEPH/features/aleph.pl...
% Warning: (/mnt/F632E52432E4EB17/ACADEMICS/5-1/Thesis-
work/ALEPH/features/aleph.pl:7211).
% Singleton variables Steam, Stream in user:p_message_file/2.
A L E P H
Version 5
Last modified: Sun Mar 11 03:25:37 UTC 2007
Manual:
http://www.comlab.ox.ac.uk/oucl/groups/machlearn/Aleph/index.html
% consulted /mnt/F632E52432E4EB17/ACADEMICS/5-1/Thesis-
work/ALEPH/features/aleph.pl in module user, 81 msec 2332976 bytes
yes
?-
```

3.4.2 Constructing a theory

Once we have all the three files (filename.b, filename.f, filename.n) with us, they can be read into Aleph with one command:

```
read_all(filename)
```

The basic command which follows the basic algorithm and constructs a hypothesis is:

```
induce.
```

When issued to Aleph, it performs the four steps described earlier. The simplest use of induce implements a simple greedy cover-set algorithm. The result is a trace that lists clauses searched along with the statistics of positive examples and negative examples it covers. For example, on running `induce` on Michalski's trains dataset, the following is part of the output.

```
eastbound(A) :-
   has_car(A,B).
[5/5]
```

```
eastbound(A) :-
   has_car(A,B), short(B).
[5/5]
eastbound(A) :-
   has_car(A,B), open_car(B).
[5/5]
eastbound(A) :-
   has_car(A,B), shape(B,rectangle).
[5/5]
```

The final result looks like:

```
[theory]

[Rule 1] [Pos cover = 5 Neg cover = 0]

eastbound(A) :-
       has_car(A,B), short(B), closed(B).

[pos-neg] [5]
```

Furthermore, `induce` also reports the accuracy on the training data by calculating and displaying a confusion matrix.

```
[Training set performance]

           Actual
        +          -
     +  5          0         5
Pred
     -  0          5         5

        5          5         10

Accuracy = 100%
```

If `test_pos` and `test_neg` parameters are set (meaning Aleph has been notified of a test database), then performance on test data is also reported using confusion matrix by Aleph.

Finally, this theory can then be saved to a file on the computer by using the following command:

```
write_rules(SomeFileName).
```

3.4.3 Advance features of Aleph

Aleph can do much more than just the basic algorithm presented above. The user is allowed to set various parameters using set(Parameter, Value) command in aleph. These parameters can then set testing files, bound the number of clauses Aleph can generate, change evaluation function for search (for finding best score), turn on lazy evaluation, turn on pretty printing, and many more things.

Among being able to change almost every setting, what makes Aleph one of the best resources for ILP researchers is its versatility. While basic Aleph algorithm is a batch learner, in the sense that all examples and background knowledge is in place before learning commences, an incremental learning mode also exists in Aleph to acquire new examples and background information by interacting with user. Aleph also allows for randomized search, learning modes, features and constraints besides clauses and offers tree-learning procedures too, as opposed to the greedy set-covering algorithm used in basic Aleph algorithm for constructing clauses.

More information on all the advance features and using them is available in The Aleph Manual [10]. A new exhaustive manual has also been prepared for Aleph version 5 as a part of this thesis.

3.5 Feature Construction in Aleph

One promising role for ILP is in the area of feature construction. A good review of the use of ILP for this can be found in S. Kramer, N. Lavrac and P. Flach (2001), *Propositionalization Approaches to Relational Data Mining*, in Relational Data Mining, S. Dzeroski and N. Lavrac (eds.), Springer. The basic Aleph algorithm constructs set of clauses. Good clauses found during the search can be used to construct boolean features, and this adds feature construction capability in Aleph. This is done by using the command `induce_features`.

By default, when user wishes to construct a theory, `induce` is executed. In case of feature construction, however, `induce_cover` is executed internally. This is generally slower and has a slightly different way of working - The positive examples covered by a clause are not removed prior to seeding on a new (uncovered) example. After a search with `induce_cover`, Aleph only removes the examples covered by the best clause are removed from a pool of seed examples only. After this, a new example or set of examples is chosen from the seeds left, and the process repeats. The theories returned by `induce` and `induce_cover` are dependent on the order in which positive examples are presented.

The process of finding rules (and the corresponding boolean features) continues till all examples are covered by the rules found or the number of features exceeds a pre-defined upper limit (which can be controlled using `set(max_features,...)`)

If we ran the feature construction using Aleph on Michalskis Trains problem, the following features are returned:

```
feature(1,(eastbound(A):-has_car(A,B),closed(B))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(2,(eastbound(A):-has_car(A,B),load(B,triangle,1))).
[pos cover = 5 neg cover = 1] [pos-neg] [4]
feature(3,(eastbound(A):-has_car(A,B),closed(B),wheels(B,2))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(4,(eastbound(A):-has_car(A,B),closed(B),has_car(A,C))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(5,(eastbound(A):-has_car(A,B),load(B,triangle,1),has_car(A,C))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(6,(eastbound(A):-has_car(A,B),has_car(A,C),closed(C))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(7,(eastbound(A):-has_car(A,B),has_car(A,C),load(C,triangle,1))).
[pos cover = 5 neg cover = 0] [pos-neg] [5]
feature(8,(eastbound(A):-has_car(A,B),short(B),closed(B))).
[pos cover = 5 neg cover = 2] [pos-neg] [3]
feature(9,(eastbound(A):-has_car(A,B),short(B),load(B,triangle,1))).
[pos cover = 4 neg cover = 1] [pos-neg] [3]
feature(10,(eastbound(A):-has_car(A,B),shape(B,rectangle),load(B,triangle,1))).
```

A feature stored by Aleph contains two bits of information:

1. **A feature identifier** : this is basically a number
2. **A clause** : Here head is a literal that unifies with any of the examples with the same name and arity as Head, and Body is a conjunction of literals. The boolean feature is marked as true if and only if an example unifies with Head and the Body is true.

Features found by Aleph can also be displayed anytime by using `show(features)` command.

Chapter 4

Maximum Entropy Model

Machine learning has ever since its origin been trying to make models that can correctly predict behaviour of a random process. For that, first thing needed is a set of statistics that capture the behaviour of the process. Secondly, given a set of statistics, the task is to corral these facts into an accurate model of the process a model capable of predicting the future output of process. The first task is famously known as feature selection and the second one is called model selection. Maximum Entropy approach is unified approach of these two steps. The maximum entropy model has proved to be significant in the field of NLP, specially for translation purposes.

The philosophy of maximum entropy modelling is: model all that is known and assume nothing about that which is unknown. In other words, given a collection of facts, choose a model which is consistent with all the facts, but otherwise as uniform as possible.

4.1 Example and Overview

We will walk through the example of experts translator decision problem presented in [15]. The problem is to construct a model that constructs proper French rendering of the English word in. Our model p of the experts decisions assigns to each French word f an estimate $p(f)$, of the probability that the expert would choose f as a translation of in.

Say we had sample data from translators and we may discover that translator always chooses among the following five french phrases : $\{dans, en, a, aucoursde, pendant\}$.

$$p(dans) + p(en) + p(a) + p(aucoursde) + p(pendant) = 1$$

This equation can then be formed and this represents our first statistic. We can now search for models that satisfy this equation. Of course, the number of satisfiable models would still be infinite. One of these could be where $p(dans) = 1$. However, clearly that would not be the correct model, since others have also appeared in the dataset collected. Maximum entropy says to not assume anything about that which is unknown. So, we go with the most uniform model, i.e.,

$$p(dans) = p(en) = p(a) = p(aucoursde) = p(pendant) = 1/5$$

Further, say more clues are obtained from the sample. For example, we notice that the expert chose either *dans* or *en* 30% of the time. This knowledge can be added to update our model of translation process by requiring p satisfy two constraints

$$\begin{gathered} p(dans) + p(en) = 3/10 \\ p(dans) + p(en) + p(a) + p(aucoursde) + p(pendant) = 1 \end{gathered}$$

Again, we will go with the most uniform model that fits these two constraints so as to not assume anything that is not known. This gives us

$$p(dans) = p(en) = 3/20; p(a) = p(aucoursde) = p(pendant) = 7/30$$

By now, it should be clear how maximum entropy is working. It can be observed that when the constraints keep on increasing, finding the most uniform model would become a difficult process and cant be done analytically. This leaves us with the questions

1. how do we measure the uniformity of the model, and
2. how do we go about finding the most uniform model subject to a set of constraints?

4.2 Mathematical Modelling

A random process is under consideration which produces an output value y. In our example, the random process generates a translation of word in, and the output y can be any word in the set $\{dans, en, a, aucoursde, pendant\}$. This set is called from now on as Y.

In generating y, the process may be influenced by some contextual information x , a member of finite set X. The task is to model a method of estimating the conditional probability that, given a context x , the process will output y. This can be denoted by $p(y|x)$, the probability that the model assigns to y in context x.

However for ease of notation, and since we'll be working with the general model rather than for each value of y and x, we denote by $p(y|x)$ the entire conditional probability distribution provided by the model. We will denote by P the set of all conditional probability distributions. Thus a model $p(y|x)$ is just an element of P.

For studying a random process, a large number of training sample $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$. is collected. In the example under discussion, each sample would consist a phrase x containing the words surrounding *in*, together with the translation y of *in* which the process produced. This initial training sample could be thought of as generated by humans or experts.

Training sample can be summarized in terms of its empirical probability distribution $\widetilde{p}$ defined by

$$\widetilde{p}(x, y) = \frac{1}{N} * \text{ number of times } (x, y) \text{ occurs in the sample}$$

The goal is then to construct a **statistical model** of the process which generated the training sample $\widetilde{p}(x, y)$.

4.3 Features and Constraints

In the example considered, the constraints that were imagined till now were independent of the context. However, in real-life scenarios, statistics should be considered that depends on contextual information x. For instance, it might be noticed that, in training sample, if *April* is the word following *in*, then the translation of *in* is *en* with frequency 9/10. To express this, we can introduce a indicator function:

$$f(x, y) = \begin{cases} 1 & \text{if } y = en \text{ and } April \text{ follows } in \\ 0 & \text{otherwise} \end{cases}$$

The expected value of f with respect to the empirical distribution $p(x, y)$ would be:

$$\widetilde{p}(f) \equiv \sum_{x,y} \widetilde{p}(x, y) f(x, y) \tag{4.1}$$

This f is called a feature function or a feature, because any statistic of the sample can now be expressed as the expected value of an appropriate binary valued indicator function f.

Whenever a statistic feels useful, it is made sure that the model p accords with it. This can be done by constraining the expected value that the model assigns to corresponding feature f. The expected value of f with respect to the model $p(y|x)$ is:

$$p(f) \equiv \sum_{x,y} \widetilde{p}(x)p(y|x)f(x,y) \tag{4.2}$$

For having our model accord with the empirical distribution of feature function, the expected value of the feature for the model is equated to the expected value of f in the training sample. This forms an equation:

$$\sum_{x,y} \widetilde{p}(x)p(y|x)f(x,y) = \sum_{x,y} \widetilde{p}(x,y)f(x,y) \tag{4.3}$$

Such an equation is called a constraint equation or simply a constraint. By restricting to those models $p(y|x)$ for which (3.3) holds, it is made sure that those models are not considered which do not agree with the training sample on how often the output of the process should exhibit the feature f. This technique can be used for every feature that seems important and useful for the model, to constraint our model $p(y|x)$ in P.

4.4 Maximum Entropy Principle

Suppose that n features are given which are important for modelling the process. The model p needs to accord with all these features. P is the set of all possible models. C is the subset of P which represents models p such that : $p(f_i) = \widetilde{p}(f_i)$, where $\widetilde{p}(f_i)$ is nothing but the probability of feature we get from the training data.

Let C_i depict a model where only f_i is satisfied (by satisfied, it means the model is in accord with that feature). This means C_1 would contain a model that satisfies only f_1. Obviously, if n features need to be satisfied, it's important to find $C_1 \cap C_2 \cap ... \cap C_n$ and that is precisely what we call C. Note that even though this is quite constrained C would still have infinite possible models. Maximum entropy philosophy states that we select the distribution/model which is most uniform.

Mathematics gives us conditional entropy as a measure of uniformity of a conditional distribution $p(y|x)$. Entropy of a model, $H(p)$, is calculated as following:

$$H(p) = -\sum_{x,y} \widetilde{p}(x)p(y|x)logp(y|x) \tag{4.4}$$

Maximum entropy principle states that to select a model from a set C of allowed probability distributions, choose a model p s.t.

$$p \in argmax(H(p)) \tag{4.5}$$

It can be shown that this p is always well defined; that is, theres always a unique model p with maximum entropy in any constrained set C.

4.5 Solving Maximum Entropy Problem

The maxium entropy principle presents us a problem of constrained optimization:

$$\textit{find } p_* \in C \textit{ which maximizes } H(p)$$

4.5.1 Dual Problem

Lagrange multipliers are used to help solve this problem. Original problem can be written as:

$$\textit{find } p_* = \operatorname*{argmax}_{p \in C} H(p)$$

For each feature f_i, a lagrange multiplier λ_i is introduced. The lagrangian $\Lambda(p, \lambda)$ is defined by:

$$\Lambda(p, \lambda) \equiv H(p) + \sum_i \lambda_i (p(f_i) - \widetilde{p}(f_i)) \tag{4.6}$$

Holding λ fixed, unconstrained maximum of the Lagrangian $\Lambda(p, \lambda)$ is computed over all $p \in P$. The model p where $\Lambda(p, \lambda)$ achieves its maximum is denoted by p_λ, and this maximum value is denoted by $\Psi(\lambda)$.

$$p_\lambda \equiv \operatorname*{argmax}_{p \in P} \Lambda(p, \lambda) \tag{4.7}$$

$$\Psi(\lambda) \equiv \Lambda(p_\lambda, \lambda) \tag{4.8}$$

$\Psi(\lambda)$ is called the dual function. The functions p_λ and $\Psi(\lambda)$ can be calculated using simple calculus and result in following formulaes:

$$p_\lambda(y|x) = \frac{1}{Z_{\lambda(x)}} \exp(\sum_i \lambda_i f_i(x, y)) \tag{4.9}$$

$$\Psi(\lambda) = -\sum_x \widetilde{p}(x) \log Z_{\lambda(x)} + \sum_i \lambda_i \widetilde{p}(f_i) \tag{4.10}$$

where $Z_{\lambda(x)}$ is a normalizing constant determined by the requirement that $\sum\limits_y p_\lambda(y|x) = 1$ for all x. The value of $Z_{\lambda(x)}$ is:

$$Z_{\lambda(x)} = \sum_y \exp(\sum_i \lambda_i f_i(x, y)) \tag{4.11}$$

At this point a dual problem can be posed. This dual problem is an unconstrained optimization problem which can be phrased as:

$$\textit{Find } \lambda^* = \operatorname*{argmax}_{\lambda} \Psi(\lambda)$$

It can be seen using Kuhn-Tucker theorem, that under suitable assumptions, the primal and dual problems are closely related. This means if λ^* is the solution of the dual problem, then p_{λ^*} is the solution of the original problem; which means $p_{\lambda^*} = p_*$.

4.5.2 Relation to Maximum Likelihood

The log-likelihood $L_{\widetilde{p}}(p)$ of the empirical distribution $\widetilde{p}$ as predicted by a model p is defined by:

$$L_{\widetilde{p}}(p) \equiv \log \prod_{x,y} p(y|x)^{\widetilde{p}(x,y)} = \sum_{x,y} \widetilde{p}(x, y) \log p(y|x) \tag{4.12}$$

It is easy to check that the dual function $\Psi(\lambda)$ of the previous section is, in fact, just the log-likelihood for the exponential model p_λ; that is

$$\Psi(\lambda) = L_{\tilde{p}}(p_\lambda) \tag{4.13}$$

What this means is, the model $p_* \in C$ with maximum entropy is the model in the parametric family $p_\lambda(y|x)$ that maximizes the likelihood of the training sample $\widetilde{p}$.

4.5.3 Parameter Computation

The λ^* that maximizes $\Psi(\lambda)$ cannot be found analytically. Fortunately, many numerical methods exist which can be used to calculate λ^*. One simple method is coordinate-wise ascent, in which λ^* is computed by iteratively maximizing $\Psi(\lambda)$ one coordinate at time. This technique is called Brown algorithm [21] when applied to maximum entropy problem. Other general purpose methods that can be used are gradient descent and conjugate gradient.

However, an optimized method specifically tailored for maximum entropy problem is Generalized Iterative Scaling(GIS) algorithm [17]. This algorithm is only applicable whenever the features $f_i(x,y)$ are non-negative, which is the case with maximum entropy problem where binary-valued feature functions are being considered. An improved version of this, called Improved Iterative Scaling(IIS) algorithm [22] generalizes the Darroch-Ratcliff procedure, which required, in addition to non-negativity of features, an extra constraint $\sum_i f_i(x,y) = 1$ for all x, y.

IIS Algorithm

1. Start with $\lambda_i = 0$ for all $i \in \{1, 2, 3, ..., n\}$
2. Do for each $\lambda_i = 0$ for all $i \in \{1, 2, 3, ..., n\}$:
   i. Let $\Delta\lambda_i$ be the solution to:

$$\sum_{x,y} \widetilde{p}(x) p(y|x) f_i(x,y) \exp(\Delta\lambda_i f^{\#}(x,y)) = \widetilde{p}(f_i) \tag{4.14}$$

$$where \quad f^{\#}(x,y)) \equiv \sum_{i=1}^{n} f_i(x,y) \tag{4.15}$$

   ii. Update the value of λ_i according to: $\lambda_i \leftarrow \lambda_i + \Delta\lambda_i$
3. Go to step 2 if not all the λ_i have converged

The key step in the algorithm is step (2a), the computation of the increments $\Delta\lambda_i$ that solve (3.14). If $f^{\#}(x,y)$ is constant for all x, y then $\Delta\lambda_i$ is given explicitly as

$$\Delta\lambda_i = \tfrac{1}{M} \log \tfrac{\widetilde{p}(f_i)}{p_\lambda(f_i)}$$

If $f^{\#}(x,y)$ is not constant, then $\Delta\lambda_i$ must be computed numerically. This can be done by Newton's method.

Chapter 5

Integrating Feature construction in Aleph and Maximum Entropy Model

5.1 Overview

As studied in Chapter 2, Aleph is known to have feature construction ability. Chapter 3 further taught that Maximum Entropy model needs a good set of features to start with, to build a model which can represent the random process accurately. The task now is to pipeline the two - use Aleph to construct features, and then feed those to Maximum Entropy Model to observe how it performs.

The long term goal with this is to construct a probabilistic model that combines statistical and logical learning. The area that does this is called Statistical Relational learning (SRL), or sometimes Probabilistic ILP. The particular kind of SRL that is being attempted in this study is of doing the logical and statistical learning separately and combining them.

5.2 A Basic implementation

Initially, by taking help of **aleph_portray** predicate of Aleph, following prolog code was added to the background knowledge file. This was done so a simple shell script would be able to output features constructed by Aleph for the dataset into a new file.

```
:- set(relation,class).
:- set(classes,[pos,neg]).
:- set(format,format(arff,[header=true,hash_features=true])).

:- set(portray_examples,true).
:- use_module(library(terms)).

aleph_portray(train_pos):-
  setting(train_pos,File),
  (setting(dependent,_) -> true; class(train_pos,Class)),
  setting(format,Format),
  show_feature_vectors(Format,File,Class).
aleph_portray(train_neg):-
  setting(train_neg,File),
  (setting(dependent,_) -> true; class(train_neg,Class)),
  setting(format,Format),
  show_feature_vectors(Format,File,Class).
aleph_portray(test_pos):-
  setting(test_pos,File),
  (setting(dependent,_) -> true; class(test_pos,Class)),
  setting(format,Format),
  show_feature_vectors(Format,File,Class).
aleph_portray(test_neg):-
```

```
  setting(test_neg,File),
  (setting(dependent,_) -> true; class(test_neg,Class)),
  setting(format,Format),
  show_feature_vectors(Format,File,Class).

class(train_pos,pos).
class(train_neg,neg).
class(test_pos,pos).
class(test_neg,neg).

show_feature_vectors(format(Format,Options),File,Class):-
  findall(N,feature(N,_),FeatureNums),
  sort(FeatureNums,FeatureList),
  (once(mem(header=true,Options)) -> show_header(Format,Options,FeatureList); true),
  open(File,read,Stream),
  repeat,
  read(Stream,Example),
  (Example = end_of_file -> close(Stream);
    show_feature_vector(Format,Options,Example,FeatureList),
    (var(Class) -> true; write(' '), write(Class), nl),
    fail),
  !.

show_header(arff,Options,FeatureList):-
  setting(relation,Relation),
  write('@relation '), write(Relation), nl,
  (once(mem(hash_features=true,Options)) -> HashFeatures=true;HashFeatures=false),
  mem(Feature,FeatureList),
  write('@attribute'), write(' '),
  write('ilp'),
  (HashFeatures = true ->
    feature(Feature,Clause),
    numbervars(Clause,0,_),
    term_hash(Clause,Hash),
    write(Hash);
    write(Feature)), tab(8),
  write('{0,1}'), nl,
  fail.
show_header(arff,_,_):-
  setting(classes,Classes),
  write('@attribute class '), tab(8),
  write('{'),
  write_classes(Classes), write('}'),
  nl, nl,
  write('@data'), nl, nl.

write_classes([Class]):-
  write(Class), !.
write_classes([Class|Classes]):-
  write(Class), write(','),
  write_classes(Classes).

show_feature_vector(arff,_,Example,FeatureList):-
  mem(Feature,FeatureList),
  feature(Feature,(Example:- Body)),
  (once(Body) -> write(1), write(', '); write(0), write(', ')),
  fail.
show_feature_vector(arff,_,Example,_):-
  setting(dependent,PredArg), !,
  arg(PredArg,Example,PredVal),
  writeq(PredVal), nl.
```

```
show_feature_vector(_,_,_,_).


writeq_list([]).
writeq_list([X|T]):-
        writeq(X), write('.'), nl,
        writeq_list(T).

mem(X,[X|_]).
mem(X,[_|T]):- mem(X,T).
```

Following shell script was written to extract features from dataset and output in a Weka compatible `.arff` file.

```
yap <<+
consult(aleph).
read_all(trains).
set(test_pos,'trains.t').
induce(features).
set(featurefile,'features.txt').
write_features.
set(verbosity,1).
show_total_stats('features_stats.txt').
stopwatch(StartClock), set(clock,StartClock).
tell('postrain.arff'), aleph_portray(train_pos), told.
stopwatch(StopClock), setting(clock,StartClock), Time is StopClock - StartClock,
    set(clock,Time).
tell('postest.arff'), aleph_portray(test_pos), told.
tell('time.txt'), setting(clock,Time), write('feature data generation time: '), write(Time),
    nl, told.
```

This features file was then converted to LIBSVM format using a simple python script following which it was fed to Maximum Entropy Modelling Toolkit for Python and C++ [16]. This constructed a Maximum Entropy model using the features and outputs a model file. This model file is then used on testing data.

5.3 Results and Observations

The maximum entropy toolkit was tested on the following datasets and the accuracies were compared against the baseline models reported in [20]. These results are presented in the table below.

Observations		
Problem	Accuracy of Presented Method (%)	Baseline Accuracy [20] (%)
Mut188	91	84.6
Canc330	55.44	50.4
DssTox	46.51	64.7
Amine	53.54	71.4
Choline	53.25	52.7
Scop	50.02	55.1
Toxic	52.73	79.2

Figure 5.1: Results on real data comparing Baseline models presented in [20] and our model

As can be observed in the table in figure 5.1, our pipelined model performs better only in 3 out of 7 problems. On one side, it has not beaten anything that old methods could do, however, at the same time, it has not been defeated totally. In Mut188, it shows an incredibly good accuracy, however, falls way behind the baseline in Amine problem. The reason for these happenings could be numerous - For one, right now all the feature constructed by Aleph are being fed to Maximum entropy model. Moreover, the greedy nature of Maximum Entropy model construction could have been the reason behind the inconsistent results.

Conclusion

The concepts of ILP and Maximum Entropy approach were carefully studied and a pipeline model was successfully developed and tested as a part of this thesis. This model utilized the features outputted by Aleph to build a Maximum Entropy model. This MaxEnt model, however, outputted a mixed bag of results, showing its precedence over the baseline model in 3 problems, and inferiority in the other 5 problems. If a coin is tossed, and out of 8 times, 3 times a head appears, it can not be said that a tail will always appear on the coin. A similar situation applies here. The mixed results only indicate that there is a need of future study and improvement in this. One area of improvement could be adding feature selection to the method - A good subset of all the features outputted by Aleph may increase the accuracy. Another way this method can undergo possible improvement is by trying combining other statistical machine learning frameworks, like SVM, or KNN, or Neural Networks with ILP.

Bibliography/References

[1] Saso Dzeroski, Nada Lavrac. *Inductive Logic Programming, Techniques and Applications.* 1994.

[2] Stephen Muggleton, Luc De Raedt. *Inductive Logic Programming: Theory and Methods.* Journal of Logic Programming 1994:19,20:629-679.

[3] Dieter Fensel, Florian Fischer. *Intelligent Systems - Inductive Logic Programming Slides.* `http://teaching-wiki.sti2.at/uploads/c/c5/11_Intelligent_Systems-InductiveLogicProgramming.pdf`

[4] E.Y.Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

[5] S.Muggleton and W.Buntine. *Machine invention of first-order predicates by inverting resolution.* In Proceedings of the Fifth International Conference on Machine Learning, pages 339-352, Kaufmann, 1988.

[6] S.Muggleton and C.Feng. *Efficient induction of logic programs.* In Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo, 1990. Ohmsha.

[7] R.Quinlan. *Learning logical definitions from relations.* Machine Learning, 5:239-266, 1990.

[8] Stephen Muggleton, CProgol, `http://www-ai.ijs.si/~ilpnet2/systems/progol.html`

[9] Rui Camacho. *Indlog - Induction in Logic* JELIA 2004 - 9th European Conference on Logics in Artificial Intelligence, editors Jose Alferes e Joao Leite, Springer-Verlag, LNAI 3229, pp 718-721, 27-30 Setembro, Lisboa, Portugal, 2004.

[10] A. Srinivasan, The Aleph Manual, `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph\_toc.html`

[11] Muggleton,S., King, R., and Sternberg, M. (1992a) *Protein secondary structure prediction using logic.* In Proc. Second International Workshop on Inductive Logic Programming. Tokyo, Japan. ICOT TM-1182.

[12] Feng, C. (1992). *Inducing temporal fault diagnostic rules from a qualitative model.* In Muggleton, S., editor, Inductive Logic Programming, pages 471-493. Academic Press, London.

[13] King, R., Muggleton, S., Lewis, R., and Sternberg, M. (1992). *Drug Design by Machine Learning : the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase.* Proc. National Academy of Sciences

[14] Muggleton, S. (1995). *Inverse Entailment and Progol.* New Generation Computing, 13, pages 245-286.

[15] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. *A Maximum Entropy Approach to Natural Language Processing.* Computational linguistics, 22(1):39-71, 1996.

[16] Maximum Entropy modelling toolkit for C++ and Python. `http://homepages.inf.ed.ac.uk/lzhang10/maxent_toolkit.html`

[17] J.N. Darroch and D. Ratcliff *Generalized iterative scaling for log-linear models.* The Annals of Mathematical Statistics, Vol. 43:pp 1470-1480, 1972.

[18] Stephen Della Pietra, Vincent J. Della Pietra, and John D. Laf ferty. *Inducing features on random fields.* IEEE transactions on Pattern Analysis and Machine Intelligence, 19(4):380-393, 1997.

[19] Joao Paulo Duarte Conceicao. *The Aleph system made easy.* University of Porto

[20] Ashwin Srinivasan and Ganesh Ramakrishnan. *Parameter screening and optimisation for ILP using designed experiments.* Journal of Machine Learning Research, 12:627-662, 2011.

[21] Brown, D. (1959) *A note on approximations to discrete probability distributions.* Information and Control, vol. 2, 386-392.

[22] Della Pietra, S., Della Pietra, V., Lafferty, J. *Inducing features of random fields.* (1995) CMU Technical Report CMU-CS-95-144.

[23] S.Kramer, N. Lavrac, and P. Flach. *Propositionalization approaches to relational data mining.* In Relational Data Mining, pages 262286. Springer-Verlag, New York, 2001.

[24] A.Srinivasan. *Excursions in Relational Feature Construction Slides.*

[25] Parikshit Sondhi. *Feature Construction Methods: A Survey.* `http://sifaka.cs.uiuc.edu/~sondhi1/survey3.pdf`

[26] Luc Dehaspe. *Maximum Entropy Modeling with Clausal Constraints.* In Proceedings of the 7th International Workshop on Inductive Logic Programming, 1997.

[27] D. Blei, A. Ng, and M. Jordan. *Latent Dirichlet allocation.* Journal of Machine Learning Research, 3:9931022, January 2003.

[28] Jun Fu Cai, Wee Sun Lee and Yee Whye Teh. *Improving Word Sense Disambiguation Using Topic Features.* In Proc. Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language learning, 2007.

[29] Timothy N. Rubin, America Chambers, Padhraic Smyth and Mark Steyvers. *Statistical topic models for multi-label document classification.* Mach Learn (2012) 88:157208.

[30] A. Srinivasan, T.A. Faruquie, I. Bhattacharya, and R.D. King. *Topic Models with Relational Features for Drug Design.* Inductive Logic Programming, Vol. 7842 (2013), pp 45-57.

[31] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning.* The MIT Press (2007).

[32] Alexandrin Popescul and Lyle H. Ungar. *Statistical Relational Learning for Link Prediction.* In Proc. of the Workshop on Learning Statistical Models from Relational Data at IJCAI-2003.

[33] Fabrizio Riguzzi, Elena Bellodi and Riccardo Zese. *A history of Probabilistic Inductive Logic Programming.* Front. Robot. AI 1:6. doi: 10.3389/frobt.2014.00006.