Nov 30 More NLP

Example: Translate the following into symbols: "Every student in CSCI2824 has passed Calculus I." Now let the domain be all students at CU.

$\forall x \quad (\text{CSCI2824}(x) \Rightarrow \text{PassedC1}(x))$

for all students

if in CSCI2824 then passed Calc I

Example: Translate the following into symbols: "Every student in CSCI2824 is either taking Data Structures, or has already passed it." (and nobody is taking it that's already passed!) Let the domain be all students at CU.

Nov 30 More NLP

Example: Translate the following into symbols: "Every student in CSCI2824 has passed Calculus I." Now let the domain be all students at CU.

Solution: for $C(x) :=$ 'x has passed Calc 1,'
and $D(x) :=$ 'x is in CSCI2824,'
we have $\forall x\,(D(x) \rightarrow C(x))$.

Example: Translate the following into symbols: "Every student in CSCI2824 is either taking Data Structures, or has already passed it." (and nobody is taking it that's already passed!) Let the domain be all students at CU.

Solution: For $D(x) :=$ 'x is in CSCI2824,'
and $DS(x) :=$ 'x is in Data Structures,'
and $P(x) :=$ 'x has passed Data Structures,'
we have $\forall x\,(D(x) \rightarrow (DS(x) \otimes P(x)))$.

Plan: post Proc #2 tomorrow
Final weekend.

Last time we learned:

1. NLP as a classifier
2. A notebook day on Active Learning (Q-learning in Python's GYM environment)

Supervised NLP

Suppose we are provided with a set of N documents that have been pre-labelled with classes, so we have a set of tuples $(document_i, class_i)$. Our goal might be to:

1. Describe the *features* (words, sentences, logical thoughts) that make a document *more likely* to belong to a specific class
2. Be able to assign a probability to a *new* or unobserved theoretical document belonging to the classes in the original corpus.

Definition: A *generative classifier* like the Naive Bayes spam filter builds a model of how a class could generate input data.

Definition: A *discriminative classifier* like the logistic regression model learns what features from the input are most useful to differentiate between classes.

Grams and Shingles

Definition: a sequence of written symbols of length N is an $N-$gram.
These can be used for e.g.:

1. Measures of *document similarity.* Comparing the frequency of grams in a set of documents allows us to compare the documents. For example, if we set N to be 20 and words, documents only "share" grams if there are *exactly copied* sequences of 20 words between them. Plagiarism, indeed!

2. A measure common for shingling and grams are *term frequency* tf and *inverse document frequency* idf. The resulting measures:

$$tf(t,d) = f_{t,d} := \text{times term t appeared in document d}$$

$$idf(t,D) := \text{measures the times term t appeared in all documents D}$$

$$\text{tf-idf}(t,d,D) = tf(t,d) \cdot idf(t,D) = f_{t,d} \cdot \frac{|D|}{count_{d \in D}(\{t \in d\})}$$

tf-idf tracks whether *important* words are observed.

Sentences and logic

In the second overview of NLP, we'll talk about the next big model: how to create better descriptions of sequences or words. In particular:

1. a gram model is often a *Markov Model*, since certain terms are much more likely to be followed by other terms. This also lets us tracking something like $P(\text{“}nofun\text{”})$ by looking at the bigrams that begin with “no.”
2. Modern AI has to read documents to figure out underlying predicate logic. For example, with a dictionary of words available, a modern AI could take the sentence “Zach likes math” and generate all kinds of *predicate logical* conclusions.

Suppose we classify “like” as a verb that needs both an object and a subject: $L(x, y) :=$ “x likes y.” Then we have statements such as
$\exists x L(x, \text{“}math\text{”})$, and $\exists y L(\text{“}Zach\text{”}, y)$,
and so forth: we can open up the whole cadre of predicate statements from discrete logic!

Logic Tools: Equivalences and Quantifiers

1. Relation by Implication: $p \rightarrow q \equiv \neg p \vee q$
2. Contraposition: $p \rightarrow q \equiv \neg q \rightarrow \neg p$
3. Biconditional: $p \Leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
4. Xor: $p \oplus q \equiv (p \vee q) \wedge \neg (p \wedge q)$

Definitions: $\forall$ is the notation "for all." $\exists$ is the notation "exists." So e.g. $\exists x P(x)$ means "there exists an x in the domain so that $P(x)$ (is true)."

DeMorgan's Laws:

1. $$\neg \forall x P(x) \equiv \exists x \neg P(x)$$
2. $$\neg \exists x P(x) \equiv \forall x \neg P(x)$$

Onto Computers

We're interested in how a computer system or database might represent sentences, people, and relations in such a way that it can:

1. Read or integrate new texts
2. Answer queries about relations within its knowledge

Definition: The *domain* of a model is the set of *domain* elements it contains. In a logical representation, the domain is typically the list of subjects and/or objects within a system or database.

We often want some simplifying assumptions to allow *database semantics* to differ slightly from an open world.

1. **Unique Names**
2. **Closed World**: sentences not already known to be true are false.
3. **Domain closure:** no objects exist outside of our knowledge.

Onto Computers

What do objects in the domain do? They *relate.*

A logical or computational representation is often based around *relations,* which are functions that take in tuples.

Example: The *parent* function might be *Parent(p,c)* and translate to: "p is the parent of c."

Example: The *child* function might be *Child(c,p)* and translate to: "c is the child of p."

In this example, the functions could be saved or stored as *inverses* of one another, as:

$$\forall\, p,\, c\, Parent(p, c) \leftrightarrow Child(c, p)$$

And we could add more:

$$\forall\, g,\, c\, Grandparent(g, c) \leftrightarrow \exists p\, Parent(g, p) \wedge Parent(p, c)$$

Defining Relations

The relations on the prior slide are the types of things that might define a family tree, or a kinship domain. Such relations are *axioms*, because they provide the basic information necessary for facts and inference in the domain.

They're also *definitions*, because they have the form of $\forall x\, y\, f(x, y) \leftrightarrow \ldots$ which create an equivalent definition or statement for the relation f.

We could also have *theorems*, which are results that flow *from* the axioms. A result of this might be that we can start with a definition of a *sibling*:

$$\forall x\, y Sibling(x, y) \leftrightarrow x \neq y \wedge \exists p Parent(p, x) \wedge Parent(p, y)$$

and *deduce* the theorem that the *Sibling* relation is symmetric:

$$\forall x\, y\, Sibling(x, y) \leftrightarrow Sibling(y, x)$$

Using Relations

One way we might use a relational database in this class is for *classification* of a state space. Suppose we have examples like our knight-seeking-treasure. The exclusivity of the state space may lead to a variety of percepts: "the square next to me is treasure, therefore it is not a dragon."

This also could lead to ways to store more complicated inferences: if the dragon could move NSEW, we might want to have a logical classifier that states: "if the 4 squares NSEW of x are not dragons, then next time step x will not be a dragon."

Computer Relations

A series of ands/ors aren't hard to code up with basic Booleans. In NLP, we want to be able to make more general statements and inferences! First this may entail setting up our database or *knowledge base*. We can TELL our knowledge system a handful of rules or relations like:

1. $\text{TELL}(database, dog(Lola))$,$\text{TELL}(database, dog(Booster))$
2. $\text{TELL}(database, Beagle(Lola))$
3. $\text{TELL}(database, \forall x Beagle(x) \implies dog(x))$

which in turn allows us to then ASK that database things like $\exists x dog(x)$ to which it might then respond "True."

If we ask for a *general* rule like "who are the dogs, x?" we get a *substitution* list $\{x/Lola\}, \{x/Booster\}$.

Computer Relations

These *substitutions* are how a system understands concepts like universal instantiation and existential instantiation.

For universal instantiation, we can substitute examples from the knowledge base into the query (if a rule holds always, it holds for specific domain elements).

This means that our computer system can take universal rules and happily substitute specific instances.

For existential instantiation, we can use examples to answer a general query (if there exists an example...)

Building a Logical Argument.

Consider the following prompt: "The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

We could train a system to read this and determine if $Criminal(West)$ is true, but it requires some prior knowledge base!

1. The words "law" and "crime" map to a single trait "criminal," a trait held by objects in the database.
2. We need to parse all the stop words, and possibly discard irrelevant statements (West is a criminal if *any* of the missiles of Nono came via them, not just *all*)

Building a Logical Argument.

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

- The law: $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \implies Criminal(x)$.
- The missiles: $Owns(Nono, M_1)$ for *some* existing $Missile(M_1)$, where $Missile(x) \implies Weapon(x)$.
- The sale: $Owns(Nono, x) \wedge Missile(x) \implies Sold(West, x, Nono)$.
- And the nations: $Enemy(Nono, America)$., $American(West)$.

Building a Logical Argument.

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

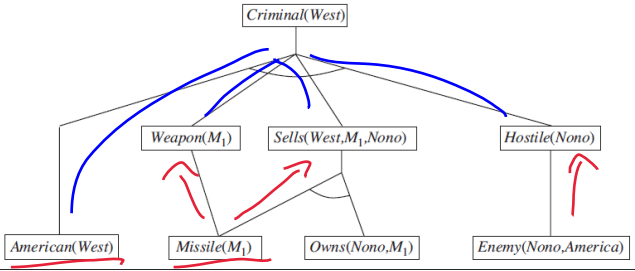

Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Chaining

The logic required for our final inference of $Criminal(West)$ is a result of the first-order (prepositional logic) *forward chaining* algorithm. In each step, we're only parsing the sentences for nouns/objects, and their verbs or adjectives (functions). After that, we glue things together using the primary rules of prepositional logic.

What's the risk of this algorithm? We only pictured the relevant conclusions. Given a large database we could have drawn tons of other possible conclusions! For example, the "Nono, an enemy of America," might trigger all sorts of inference in our model about the foreign policies of the two countries, which ultimately never helped with the *Criminal* inference.

Backwards Chaining

An alternative to figuring our whether an ASK prompt is true is *backward chaining*: what are the minimum conditions necessary for the sentence to be true? From there, we can try to *back out* the truth value of the query.

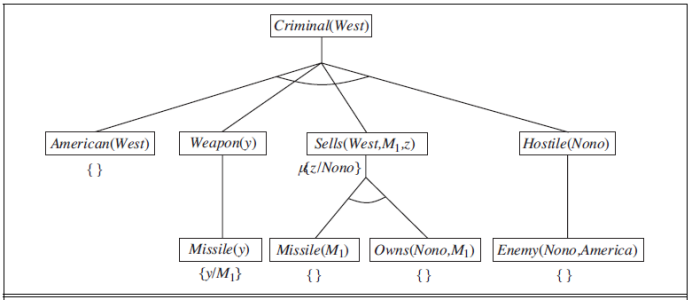

Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

Logical Form

Of course, the prior of a system that can actually *read* lies in its ability to decompose actual sentences into connected logicals. For that, we do have a nice result:

Theoerem: Every sentence of first order logical can be converted into an inferentially equivalent conjunctive normal form sentence. *Conjunctive normal form* writes the sentences as a conjunction of clauses: or a list of *and/or* statements connecting literals (assertions on elements or variables).

Example: The desired inference of $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \implies Criminal(x)$ can be rewritten as "ands" using relation by implication (recall: $p \implies q \leftrightarrow \neg p \vee q$). It is equivalent to:

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$

Logical Form

The reduction of a complex query into conjunctive normal form allows a computer to answer it with *only* queries of true/false ASKS, since those can quickly resolve any and or or statement.

Example: Resolve the following query:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

Resolution

In logic, *resolution* was our statement to infer the process of elimination:

$$[(p \vee q) \wedge \neg q] \implies p$$

Writing logical statements into ands and ors allows a query to be answered by considering the proposition "is the system *satisfiable* if Curiosity killed Tuna?" Alternatively, "is the system *satisfiable* if Curiosity did not kill Tuna?"

If we want to prove that Curiosity *did* kill Tuna, we would start with the *assumption* that Curiosity *did not* kill Tuna and then derive a contradiction. This is known as the **resolution** algorithm for logical inference.

Resolution

In logic, *resolution* was our statement to infer the process of elimination:

$$[(p \vee q) \wedge \neg q] \implies p$$

Writing logical statements into ands and ors allows a query to be answered by considering the proposition "is the system *satisfiable* if Curiosity killed Tuna?" Alternatively, "is the system *satisfiable* if Curiosity did not kill Tuna?"

If we want to prove that Curiosity *did* kill Tuna, we would start with the *assumption* that Curiosity *did not* kill Tuna and then derive a contradiction. This is known as the **resolution** algorithm for logical inference. So... did Curiosity kill the cat?

Resolution

In logic, *resolution* was our statement to infer the process of elimination:

$$[(p \vee q) \wedge \neg q] \implies p$$

Writing logical statements into ands and ors allows a query to be answered by considering the proposition "is the system *satisfiable* if Curiosity killed Tuna?" Alternatively, "is the system *satisfiable* if Curiosity did not kill Tuna?"

If we want to prove that Curiosity *did* kill Tuna, we would start with the *assumption* that Curiosity *did not* kill Tuna and then derive a contradiction. This is known as the **resolution** algorithm for logical inference. So... did Curiosity kill the cat?

Suppose Curiosity did not kill Tuna. We know that either Jack or Curiosity did; thus Jack must have. Now, Tuna is a cat and cats are animals, so Tuna is an animal. Because anyone who kills an animal is loved by no one, we know that no one loves Jack. On the other hand, Jack loves all animals, so someone loves him; so we have a contradiction. Therefore, Curiosity killed the cat.

In the text

For discussion of implementation, efficiency, etc. you can find more examples in chapter 7-9 of Norvig. These three algorithms:

1. Forward-chaining logic
2. Backward-chaining logic
3. Resolution

provide an introduction to the ways in which a knowledge base or AI can take smaller sets of information and answer more complicated queries. We haven't touched on the next major issue, which is how to teach an AI to correctly parse prose for the appropriate conjunctives or logicals. For this, we likely need access to a *dictionary* wherein the computer understands which terms are the verbs/adjectives for relations and which words are the nouns to be subjects/objects of those relations.

Extraction

In chapter 22, Norvig covers some techniques in *extraction*. We provide our reader with a background set of groupings. The FASTUS model classifies words in a sentence as:

1. NG: noun group
2. VG: verb group
3. CJ: conjunctions such as and/or **or** logicals that can be rewritten as conjunctions
4. PR: prepositions

Such a classification is *finite-state*, and probably won't pick up nuance very well. But it can start to understand texts!

Extraction

So we feed a test-reader a sentence or thought, and it will:

1. Tokenize the characters into tokens such as words, punctuation, numbers. This may be the same as shingling on words (1-grams).
2. Handle complex words: group tokens like Firstname Lastname, "set up," "exploratory analysis" into *single* NG or VG objects.
3. Handle basic groups: assign the NG/VG/CJ/PR statuses.
4. Handle complex phrases: create or classify NG/VG pairs, creating the relations like $loves(Beagles, Zach)$.
5. Merge Structures: use additional sentences or prepositions to glue together sequential and related thoughts.

Noise and Extraction

The model outlined before is a simple finite-state model approach, and may not work well on varied or noisy inputs. More complex models encode language as a Markov model or HMM. One task might be to create a language HMM that to tries to read a course description to figure out how much math is needed.

1. The HMM may look for sequences of words such as "requires (a background in)..." and then a set of trained keywords.
2. Words like "familiarity," "rigor," "proof" adjust the underlying probabilities of how much knowledge we might need.

Moving Forward

- Coming up:
  1. We talk more AI classifiers! Perceptrons, and NNs!