Medicare Fraud Detection using CLIPS

By Alex Rodgers, Jackson Falin, and Josh Grace

1- Project description: main idea, features, how it's related to AI

Our project is to detect Medicare providers, such as hospitals or clinics, that create fraudulent claims Medicare claims using a rules-based system. We wanted to compare our expert system to an approach with the same data done by a Neural Network. Our program allows a user to take CSV files containing Medicare claims and identify which providers are engaged in fraudulent activities. Our program features a parser to translate CSV files into CLIPS information, CLIPS rules which can detect provider fraud, and a testing script to evaluate our detections against confirmed cases of fraud. Our project is related to AI because it takes existing information (CSV files of Medicare claim data), and using predefined rules, generate additional information about the data set (what providers are engaged in Medicare fraud).

2- Project design: general architecture, components, existing components/APIs.

For our data, we used a Kaggle dataset containing Medicare claims. This dataset included four CSV files describing the claims. They had beneficiary(patient) information files, which included Birth/ Death date, chronic conditions, etc. There were also Outpatient and Inpatient files that described patients’ visits to the hospital/ clinics, including what they were treated for, what doctors they saw, and what they were treated for, etc. Finally, there was a CSV file the described what providers had been flagged for fraud. We used CLIPS, an expert system that enables the development of rules-based as our detection system. First, we parsed our insurance claim CSV files into files that CLIPS could read. Next, we loaded our rules into CLIPS. These defined the attributes of an insurance claim that may indicate fraud. Finally, we ran our facts through the rules. After the rules ran, we had an additional set of rules that decided if a specific provider had enough instances of potential fraud to be flagged as fraudulent. When a provider is flagged as fraudulent, it’s name is printed to an external file. Once CLIPS was finished running, an additional Python script compared the output file to the list of providers marked and outputted the success metrics of our script.

CLIPS templates: To allow CLIPS to understand our facts, we first had to write template files for each of the 4 CSV files. These templates define what information is contained in each field of the facts. This gives information to CLIPS about where the elements of an insurance claim are located in each fact. These templates allow our rules to work with specific aspects of each insurance claim, making our rules more human-readable and simpler to develop.

Parser: The first step was to translate our CSV files with insurance claims information into CLIPS facts. CLIPS requires knowledge to be in a specific fact format to be read by the system. So, we wrote a Python parser which read through our CSV files and translated the information into fact files that could be read by CLIPS. The parser reads each CSV field for each Medicare claim and puts the information into the corresponding fact field as defined by the template.

Rules: PUT RULES HERE

Testing Script: Our testing script was the final step in our workflow. As CLIPS ran, it places any providers the contained fraud into a temporary file. When our testing script started, it first read a text file containing all the providers that were marked in the dataset as containing fraud. It then placed those provider names into a dictionary. Then, the script read through the output file from CLIPS, and if any providers included in the output script also appeared in the dictionary, then the script prints all true positives from the flagged providers.

Overall Architecture: To automate running CLIPS, we wrote bash scripts to enable faster development. To set up our CLIPS facts, we ran csvToClipsFacts.py and csvBeneficiaryToClipsFacts.py. These scripts are parsers that translate our CSV files into CLIPS facts. To run our program, all we had to do is run ./startClipsFacts.sh. This script pipes our commands (stored in startClipsCommands.txt) into CLIPS’ stdin, and once CLIPS is finished running, the script runs the evaluation Python code (parseClipsOutput.py) and removes any temporary files. startClipsCommands.txt contains the code that loads the template files, fact files, and rules files, and the code to run the facts through CLIPS.

3- Implementation: language, frameworks, flow diagrams

Language: We wrote all our external scripts in Bash or Python. All CLIPS specific code/ data was written in a CLIPS specific format.

Frameworks: Our only framework outside of standard Python libraries was CLIPS. CLIPS was used to run our Medicare claim information from Kaggle against predefined rules to detect provider fraud.

Flow Diagram: A screenshot of a cell phone

Description automatically generated

4- Testing: test cases covering all features, results

Our dataset was far too large to permit manually read through and flag providers as generating fraud. Fortunately, the Kaggle dataset came with a list of INSERT PROVIDER TYPE HERE providers that had previously been marked for fraud. As such, our testing script parsed out the providers that were marked for fraud and compared the truly fraudulent providers with the flagged providers. Further complicating manually flagging data, Medicare claims are so diverse, there isn’t a single technique used to identify provider fraud, so we must rely on the previously flagged data to give us a benchmark for how well our detections work with a specific subset of the data, and extrapolate our system’s performance from this data. Based on the number of providers flagged, we can assume that the marked providers cover a variety of fraud types. INSERT RESULTS HERE

5- Analysis: discussion of project, difficulties with project (research, platforms, dealing with existing code, external components performance, etc.) and suggestions for extensions.

To develop our system, we began by selecting a dataset. We were looking for data that was from an interesting source, that also had enough fields/ information about claims that we could do some interesting analysis on the data. After looking at a few different insurance fraud data sets, we decided to use the Medicare fraud data set.

Initially, we ran into trouble deciding how to represent the data. The Medicare set had eight separate CSV files (Beneficiary, Outpatient, Inpatient, and flagged providers, each with testing/ training data), each containing lots of information. So, we needed a good way to correlate those data sets while maintaining each data type separately. After talking to Dr. Assal, we elected to use different templates for each of the CSV files (beneficiary, outpatient, inpatient), and an additional Provider template containing the name and number of instances of fraud per provider. We also store the flagged providers in an external .txt file that never interacts with CLIPS to ensure that our system isn’t cheating. The Provider template allows us to correlate the Outpatient and Inpatient claims to their respective provider, and at the end of our program, read through the provider information and flag potential fraud. In hindsight, we think that this was the best way to architect our information. This approach allowed us to write a relatively simple parser while also allowing our data to be correlated across fact files so we can do reasoning across specific data files.

Our next challenge was developing the rules TALK ABOUT ISSUES CREATING RULES

For extensions, we’d be very interested in researching how our rules work on additional Medicare data, or Medicare data from a different period. We’re especially interested to see if the fraud methods are evolving, as the government detects/ prosecutes providers engaging in fraudulent activities. We are also interested in seeing how well our process for detecting fraud works for other types of insurance fraud like Car or house insurance fraud.

6- Comparing a CLIPS approach to a Neural Network approach

INSERT RESULTS HERE