**Structured Abstract**

**Background/Context**:

Artificial neural networks (ANN) are computing systems that are inspired by, but not identical to, biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules. An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. For our problem domain we needed to design a system that automatically classified a set of non-functional requirements into different categories non-functional requirements.

**Problem Definition:**

The overall research questions our group addressed is how to teach a program to recognize and learn information from the given data set. First was how to design and algorithm that produces a score rather than a probability. We want it to ignore predictions below some defined threshold. Next is the algorithm need to ‘learn’ from examples for what is in a class from the given data set, but not what isn’t. Both questions where steps to solve the larger problem with completing this project.

**Approach:**

Our group took the bag of words approach to solve how a machine can learn from the training data. This is a way of extracting features from text for use in modeling, such as with machine learning algorithms. The approach is very simple and flexible and can be used in a myriad of ways for extracting features from documents.

**Expected Results:**

The process of creating and teaching a neural network is a tedious process, but any but of research and implementation can contribute to the field on how the software system is developed. In this project we present a simple effort to make an ANN work, our approach is like another, but may help analysts, architects, and developers save unnecessary manual analysis.

**Conclusions:**

The research of applying ANN techniques into automatic requirement analysis is continuing. There are several ways for future research and improvements to these systems. We want to design and build these systems in a novel approach to accommodate the need od a fully automated, comprehensive, and machine learning-based requirement analysis and software development system.

**Data set**

**Description:**

Our Training data was a text-based set of data that was either to be classified as function or non-function requirements.

**Data Pre-processing steps:**

First to clear up the data set we had to come up with a clear definition of what is functional and what is nonfunctional. First, we defined the terms:

The definition of a functional requirement is - Any requirement which specifies what the system should do.

Typical functional requirements include:

* Business Rules
* Transaction corrections, adjustments and cancellations
* Administrative functions
* Authentication
* Authorization levels
* Audit Tracking
* External Interfaces
* Certification Requirements
* Reporting Requirements
* Historical Data
* Legal or Regulatory Requirements

The definition of a non-functional requirement is - Any requirement which specifies how the system performs a certain function.

Performance – for example: response time, throughput, utilization, static volumetric

* Scalability
* Capacity
* Availability
* Reliability
* Recoverability
* Maintainability
* Serviceability
* Security
* Regulatory
* Manageability
* Environmental
* Data Integrity
* Usability
* Interoperability

After clearly defining these two terms we where able to clean up the data set and organize them into separate text files.

**Solution Methodology**

**Description of the model:**

For our model we chose the bag of words approach. This model seemed like the most common used and most efficient.

**Model code and source:**

We begin by importing our natural language toolkit. We need a way to reliably tokenize sentences into words and a way to stem words:

# use natural language toolkit

import nltk

​

nltk.download('punkt')

from nltk.stem.lancaster import LancasterStemmer

import os

import json

import datetime

import numpy as np

import time

**And our training data:**

training\_data = [

{"class": "security", "sentence": "A session that has been inactive for more than ten minutes is terminated "},

{"class": "security", "sentence": "- Disable accounts after 90 days of inactivity"},

{"class": "security", "sentence": "- Passwords can be changed by the associated user only once in a 2-day period"},

{"class": "security", "sentence": "- Security decisions are not made based on user-supplied file names and paths."},

{"class": "privacy",

"sentence": "(1) An authorization or other express legal permission from an individual to use or disclose protected health information for the research;"},

{"class": "privacy",

"sentence": "(1) An individual has a right to receive an accounting of disclosures of protected health information made by a covered entity in the six years prior to the date on which the accounting is requested, except for disclosures:"},

{"class": "privacy",

"sentence": "(1) Not use or further disclose the information other than as permitted by the data use agreement or as otherwise required by law;"},

{"class": "privacy",

"sentence": "(1) To conduct an evaluation relating to medical surveillance of the workplace; or"}

]

**Data structures for documents, classes and words:**

# security data

# privacy data

words = []

classes = []

documents = []

ignore\_words = ['.', ';', ',', '-', ':']

for pattern in training\_data:

w = nltk.word\_tokenize(pattern['sentence']) # tokenizes words in the sentence

words.extend(w) # add to a word list

documents.append((w, pattern['class'])) # add to documents

if pattern['class'] not in classes: # add to classes list (privacy and security)

classes.append(pattern['class'])

words = [stemmer.stem(w.lower()) for w in words if

w not in ignore\_words] # stems(playing --> play) and lowercases each word(SONG --> song)

words = list(set(words))

classes = list(set(classes)) # removes dupes

print(len(documents), "documents")

print(len(classes), "classes", classes)

print(len(words), "unique stem words", words

**Our training data is transformed into “bag of words” for each sentence:**

training = []

output = []

output\_empty = [0] \* len(classes) # empty array for outputs

for doc in documents: # training set bag of words

bag = [] # initializes the bag array

pattern\_words = doc[0] # list of tokenized words

pattern\_words = [stemmer.stem(word.lower()) for word in pattern\_words] # stems and lowercases each word

for w in words: # fills array up with bag of words

bag.append(1) if w in pattern\_words else bag.append(0)

​

training.append(bag)

output\_row = list(output\_empty)

output\_row[classes.index(doc[1])] = 1

output.append(output\_row)

​

i = 0

w = documents[i][0]

print([stemmer.stem(word.lower()) for word in w])

print(training[i])

print(output[i])

**Our core functions for our 2-layer neural network. We use a sigmoid function to normalize values and its derivative to measure the error rate. Iterating and adjusting until our error rate is acceptably low:**

​

# compute sigmoid

def sigmoid(x):

output = 1 / (1 + np.exp(-x))

return output

​

​

# covert output of sigmoid function to it derivative

def sigmoid\_output\_to\_derivative(output):

return output \* (1 - output)

​

​

def clean\_up\_sentence(sentence):

# tokenize the pattern

sentence\_words = nltk.word\_tokenize(sentence)

# stem each word

sentence\_words = [stemmer.stem(word.lower()) for word in sentence\_words]

return sentence\_words

​

​

def bow(sentence, words, show\_details=False):

# tokenize the patter

sentence\_words = clean\_up\_sentence(sentence)

# bag of words

bag = [0] \* len(words)

for s in sentence\_words:

for i, w in enumerate(words):

if w == s:

bag[i] = 1

if show\_details:

print("found in bag: %s" % w)

return np.array(bag)

​

​

def think(sentence, show\_details=False):

x = bow(sentence.lower(), words, show\_details)

if show\_details:

print("sentence:", sentence, "\n bow:", x)

# input layer is our bag of words

l0 = x

# matrix mulitplication of input and hidden layer

l1 = sigmoid(np.dot(l0, synapse\_0))

# outerlayer

l2 = sigmoid(np.dot(l1, synapse\_1))

return l2

​

​

def train(X, y, hidden\_neurons=10, alpha=1, epochs=50000, dropout=False, dropout\_percent=0.5):

print("Training with %s neurons, alpha:%s, dropout:%s %s" % (

hidden\_neurons, str(alpha), dropout, dropout\_percent if dropout else '')

)

print("Input matrix: %sx%s Output matrix: %sx%s" % (

len(X), len(X[0]), 1, len(classes))

)

np.random.seed(1)

​

last\_mean\_error = 1

​

# randomly initialize out weights with mean 0

synapse\_0 = 2 \* np.random.random((len(X[0]), hidden\_neurons)) - 1

synapse\_1 = 2 \* np.random.random((hidden\_neurons, len(classes))) - 1

​

prev\_synapse\_0\_weight\_update = np.zeros\_like(synapse\_0)

prev\_synapse\_1\_weight\_update = np.zeros\_like(synapse\_1)

​

synapse\_0\_direction\_count = np.zeros\_like(synapse\_0)

synapse\_1\_direction\_count = np.zeros\_like(synapse\_1)

​

for j in iter(range(epochs + 1)):

​

# Feed forward through layers 0, 1, and 2

layer\_0 = X

layer\_1 = sigmoid(np.dot(layer\_0, synapse\_0))

​

if dropout:

layer\_1 \*= np.random.binomial([np.ones((len(X), hidden\_neurons))], 1 - dropout\_percent)[0] \* (

1.0 / (1 - dropout\_percent))

​

layer\_2 = sigmoid(np.dot(layer\_1, synapse\_1))

layer\_2\_error = y - layer\_2

# if this 10k iteration's error is greater than the last iteration, break out

if (j % 10000) == 0 and j > 5000:

if np.mean(np.abs(layer\_2\_error)) < last\_mean\_error:

print("delta after " + str(j) + " iterations:" + str(np.mean(np.abs(layer\_2\_error))))

last\_mean\_error = np.mean(np.abs(layer\_2\_error))

else:

print("break:", np.mean(np.abs(layer\_2\_error)), ">", last\_mean\_error)

break

​

# in what direction is the target value?

# were we really sure? if so, don't change too much.

layer\_2\_delta = layer\_2\_error \* sigmoid\_output\_to\_derivative(layer\_2)

​

# how much did each l1 value contribute to the l2 error (according to the weights

layer\_1\_error = layer\_2\_delta.dot(synapse\_1.T)

​

# in what direction is the target l1?

# were we really sure? is so, don't change too much

layer\_1\_delta = layer\_1\_error \* sigmoid\_output\_to\_derivative(layer\_1)

​

synapse\_1\_weight\_update = (layer\_1.T.dot(layer\_2\_delta))

synapse\_0\_weight\_update = (layer\_0.T.dot(layer\_1\_delta))

​

prev\_synapse\_0\_weight\_update = synapse\_0\_weight\_update

prev\_synapse\_1\_weight\_update = synapse\_1\_weight\_update

​

if j > 0:

synapse\_0\_direction\_count += np.abs(

((synapse\_0\_weight\_update > 0) + 0) - ((prev\_synapse\_0\_weight\_update > 0) + 0))

synapse\_1\_direction\_count += np.abs(

((synapse\_1\_weight\_update > 0) + 0) - ((prev\_synapse\_1\_weight\_update > 0) + 0))

​

synapse\_1 += alpha \* synapse\_1\_weight\_update

synapse\_0 += alpha \* synapse\_0\_weight\_update

​

now = datetime.datetime.now()

​

# persist synapses

synapse = {'synapse0': synapse\_0.tolist(), 'synapse1': synapse\_1.tolist(),

'datetime': now.strftime("%Y-%m-%d %H:%M"),

'words': words,

'classes': classes

}

synapse\_file = "synapses.json"

​

with open(synapse\_file, 'w') as outfile:

json.dump(synapse, outfile, 4, True)

print("saved synapses to:", synapse\_file)

**Our neural network model:**

​

X = np.array(training)

y = np.array(output)

​

start\_time = time.time()

​

train(X, y, 20, 0.1, 100000, False, 0.2)

​

elapsed\_time = time.time() - start\_time

print("Processing time:", elapsed\_time, "seconds")

​

​

ERROR\_THRESHOLD = 0.2

**Description of Model run**

**Conclusion:**