ECE 460J Data Science Lab 2 Report:

Authors: William Wooten, Aksheeth Ilamparithi

Department of Electrical and Computer Engineering, The University of Texas at Austin

Problem 1

‘A Mathematical Theory of Communication’, authored by Claude E. Shannon and published in 1948, gives us a means to define mathematical models that correspond to the historical transmission of data. That is, Shannon seems to be detailing what came to be modern information theory. In general, a ‘communication system’ boils down to these essential 5 parts: the information source, the transmitter, the channel, the receiver, and the destination. For the information source, Shannon defines the “stochastic process”: existing mathematical models that depend on preset conditions can be leveraged to make numerous predictions of some deterministic probability. The “Markov process” is also notably highlighted; this process is a graphic representation of the information source defined by transitioning states where each state of the info is shown to have a preceding state as well as a number of potential succeeding states it can transition to. When encoding info, we pay the most attention to the relative entropy, the maximum rate of compression we can employ when encoding data into alphanumeric characters. Once the information source performs its duty in making a prediction, a transducer then encodes/decodes this produced message. Our ultimate goal, defined by Shannon, is to find the most ‘efficient’ encoding of information as it travels across a channel, and this is determined by 2 main things: the capacity in bits per second, which we aim to maximize, and entry in bits per symbol of the source, which we aim to minimize.

Problem 2

To understand what the top 10 words were from the documents, we had to scrape and compile all the pdfs contents into a single string. Here is how that was accomplished.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(req.content, 'html.parser')
# for each div corresponding to a paper
pdf_links = []
for div_paper in div_wrapper.find_all('div', class_ = 'paper'):
```

```
    links = div_paper.find('p', class_ = "links")
    # Get the pdf
    a = links.find_all('a')
    # Save the links ending in .pdf
    for content in a:
        # Get the link
        href = content.get('href')
        # Check if it is a pdf
        if href.find('.pdf') != -1:
            pdf_links.append(href)
import PyPDF2 as ppdf
from io import BytesIO
giant_string = ""
count = 0
pdfs = []
for pdf_link in pdf_links:
    with requests.get(pdf_link) as req:
        with BytesIO(req.content) as data:
            try:
                read_pdf = ppdf.PdfFileReader(data)
                for page in range(read_pdf.getNumPages()):
                    giant_string += " " + read_pdf.getPage(page).extractText()
                pdfs.append(read_pdf)
            except:
                print(data)
    count += 1
#Some links return a 404 error thus creating an exception when trying to read a pdf.
```

Next we use the Counter library from Collections to number the 10 most common words

```
from collections import Counter
giant_string = giant_string.replace("\n", " ")
split_it = giant_string.split()

counter = Counter(split_it)

top_10_words = counter.most_common(10)
top_10_words
```

With output:

```
[('the', 2928),
 ('of', 1699),
 ('and', 941),
 ('to', 823),
 ('a', 784),
```

```
 ('is', 671),
 ('in', 645),
 ('for', 638),
 ('that', 582),
 ('we', 428)]
```

These are the 10 most frequently used words from the scraped pdfs. Counter is a dictionary-like object mapping words to their frequencies, which we will use later to estimate pmfs.

Here is the code to estimate the entropy of random variable Z:

```
# Create a distribution for choosing a random word from a random paper, i.e. choose
a random paper then a random word
Z = []
for pdf in pdfs:
    curr_paper = ""
    for page in range(pdf.getNumPages()):
        curr_paper += " " + pdf.getPage(page).extractText()
    curr_paper = curr_paper.replace("\n", " ").split()
    word_count = Counter(split_it)
    for word in word_count:
        word_count[word] /= (len(curr_paper) * len(pdfs))
        Z.append(word_count[word])
entropy(Z, base = 2)
```

Entropy differs when we limit the pdfs used, but is approximately 17.

Lastly, to synthesize a random paragraph we used the following code. Notice ho we removed certain characteristics, which indicate that a “Word” is not actually a word.

```
# Make a Dataframe from the words
counts = {}
in_order =  counter.most_common(len(counter))
total_words = 0
for pair in in_order:
    counts[pair[0]] = [pair[1]]
    total_words += pair[1]
import pandas as pd
df = pd.DataFrame(counts)
df.head()
```

```
# Remove every word that has only one occurrence
rem = []
for word in df:
    if df[word][0] == 1 or "," in word or ")" in word or "(" in word:
        rem.append(word)
df = df.drop(labels = [word for word in rem],axis = 1)

#In lecture, prof. said to remove all words with frequency one, likely getting rid
of words that #arent actually words (equations).

for word in df:
    num = df[word][0]  #DF[COLUMN][ROW]
    df[word] = num / (total_words)
word_probs = df.iloc[0].to_numpy() #an array of probabilities
df.head()
```

```
import random
words = df.columns
random_words_paragraph = random.choices(words, weights = word_probs, k = 50)
for word in random_words_paragraph:
    print(word, end = " ")
```

This is the random paragraph generated from the algorithm:

```
R. discarded shows where many node seen in search and Define
exponential a and this FTPL the is of The to but to iterateπ points
slowly of we network seen to in Binomial that balanced the partially
for the this constraints.
```

Problem 3

(a)

The best Kaggle forum post for me personally was “Regularized Linear Models.” In my opinion, this gave me the best intro to the models we initiated in class- Lasso, Ridge, and XGB. Building off of these regularization models, I learned the most important hyperparameters for each, including the learning rate for boosts, n estimator/depths for lasso and ridge, and alpha, the regularization parameter. We learned the relationship between these parameters and our ultimate Kaggle score. For instance, if we make alpha 0, we are heavily overfitting the model as we are using only raw data and no regularization, leading to a very strong score on the training set but a

terrible score on the test data. Prior to these steps, we also expanded our data pre-processing toolbox including numerous ways to process categorical features and to handle skewed data.

(b)

The best public leaderboard score we could achieve before the deadline was 0.124. We used feature engineering, and parameterization. Given more time we would have optimized the parameters further. For feature engineering we tried the following tactics:

1 - Remove significant outliers for highly correlated features

2 - Replace NaNs with 0 for numerical features that correspond to something missing (e.g pool size)

3 - Replace NaNs with none for categorical features that correspond to something missing (e.g. pool quality)

4 - replace NaN values with intelligent means and average (replace lot size when missing with the average for the area it is in)

We also transformed skewed data using boxcox1p() transformation.

(c)

As shown in the code, we found that we can vastly overfit a model by making our n-estimators and depth of the decision tree abnormally large. We also made alpha low to minimize regularization. On the contrary, underfitting was accomplished via heavy regularization and small tree depth.