# OPTION #2: NLP Chatbot Final Version

Scott Miner

Colorado State University – Global Campus

# Abstract

```
class chatbot:
    def __init__(self):
        self.max_vocab_size = 50000
        self.max_seq_len = 30
        self.embedding_dim = 100
        self.hidden_state_dim = 100
        self.epochs = 80
        self.batch_size = 128
        self.learning_rate = 1e-4
        self.dropout = 0.3
        self.data_path = r'G:\My Drive\chatbot\twcs.csv'
        self.outpath = seq2seq_path
        self.version = 'v1'
        self.mode = 'inference'
        self.num_train_records = 50000
        self.load_model_from = os.path.join(seq2seq_path, 's2s_model_v1_.h5')
        self.vocabulary_path = os.path.join(seq2seq_path, 'vocabulary.pkl')
        self.reverse_vocabulary_path = os.path.join(seq2seq_path, 'reverse_vocabulary.pkl')
        self.count_vectorizer_path = os.path.join(seq2seq_path, 'count_vectorizer.pkl')
        self.UNK = 0
        self.PAD = 1
        self.START = 2

        # intent model variables
        self.intent_load_model_from = os.path.join(intents_path, 'intents_chatbot_model.h5')
        self.intent_load_intents_from = os.path.join(intents_path, 'intents_job_intents.json')
        self.intent_load_classes = os.path.join(intents_path, 'intents_classes.pkl')
        self.intent_load_words = os.path.join(intents_path, 'intents_words.pkl')

    def process_data(self, path):
        data = pd.read_csv(path)
        if self.mode =='train':
            data = pd.read_csv(path)
            data['in_response_to_tweet_id'].fillna(-12345, inplace=True)
            tweets_in = data[data['in_response_to_tweet_id'] == -12345]
            tweets_in_out = tweets_in.merge(data, left_on=['tweet_id'], right_on=['in_response_to_tweet_id'])
            return tweets_in_out[:self.num_train_records]
        elif self.mode == 'inference':
            return data
```

*Figure 1*. The chatbot's constructor and its hyperparameters

```
+ Code + Text                                                                    Connect

def define_model(self):

    # Embedding Layer
    embedding = Embedding(
        output_dim=self.embedding_dim,
        input_dim=self.max_vocab_size,
        input_length=self.max_seq_len,
        name='embedding',
    )
    # Encoder input
    encoder_input = Input(
        shape=(self.max_seq_len,),
        dtype='int32',
        name='encoder_input',
    )
    embedded_input = embedding(encoder_input)

    encoder_rnn = LSTM(
        self.hidden_state_dim,
        name='encoder',
        dropout=self.dropout
    )

    # Context is repeated to the max sequence length so that the same context
    # can be feed at each step of decoder
    context = RepeatVector(self.max_seq_len)(encoder_rnn(embedded_input))

    # Decoder
    last_word_input = Input(
        shape=(self.max_seq_len,),
        dtype='int32',
        name='last_word_input',
    )

    embedded_last_word = embedding(last_word_input)
    # Combines the context produced by the encoder and the last word uttered as inputs
    # to the decoder.

    decoder_input = concatenate([embedded_last_word, context], axis=2)

    # return_sequences causes LSTM to produce one output per timestep instead of one at the
    # end of the input, which is important for sequence producing models.
    decoder_rnn = LSTM(
        self.hidden_state_dim,
        name='decoder',
        return_sequences=True,
        dropout=self.dropout
    )

    decoder_output = decoder_rnn(decoder_input)

    #TimeDistributed allows the dense layer to be applied to each decoder output per timestep
    next_word_dense = TimeDistributed(
        Dense(int(self.max_vocab_size / 20), activation='relu'),
        name='next_word_dense',
    )(decoder_output)

    next_word = TimeDistributed(
        Dense(self.max_vocab_size, activation='softmax'),
        name='next_word_softmax'
    )(next_word_dense)

    return Model(inputs=[encoder_input, last_word_input], outputs=[next_word])

def create_model(self):
    _model_ = self.define_model()
    adam = Adam(learning_rate=self.learning_rate, clipvalue=5.0)
    _model_.compile(optimizer=adam, loss='sparse_categorical_crossentropy')
    return _model_

# Function to append the START indext to the response Y
def include_start_token(self, Y):
    print(Y.shape)
    Y = Y.reshape((Y.shape[0], Y.shape[1]))
    Y = np.hstack((self.START * np.ones((Y.shape[0], 1)), Y[:, :-1]))
    # Y = Y[:,:,np.newaxis]
    return Y

def binarize_output_response(self, Y):
    return np.array([np_utils.to_categorical(row, num_classes=self.max_vocab_size)
                     for row in Y])

def respond_to_input(self, model, input_sent):
    input_y = self.include_start_token(self.PAD *np.ones((1, self.max_seq_len)))
    ids = np.array(self.words_to_indices(input_sent)).reshape((1, self.max_seq_len))
    for pos in range(self.max_seq_len - 1):
        pred = model.predict([ids, input_y]).argmax(axis=2)[0]
        # pred = model.predict([ids, input_y])[0]
        input_y[:, pos + 1] = pred[pos]
    return self.indices_to_words(model.predict([ids, input_y]).argmax(axis=2)[0])

def train_model(self, model, X_train, X_test, y_train, y_test):
    input_y_train = self.include_start_token(y_train)
```

*Figure 2.* Sequence-to-sequence (Seq2Seq) model

```
DataFrame shape: (2811774, 7)
Rows: 2811774
Cols: 7
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2811774 entries, 0 to 2811773
Data columns (total 7 columns):
 #   Column                  Dtype
---  ------                  -----
 0   tweet_id                int64
 1   author_id               object
 2   inbound                 bool
 3   created_at              object
 4   text                    object
 5   response_tweet_id       object
 6   in_response_to_tweet_id  float64
dtypes: bool(1), float64(1), int64(1), object(4)
memory usage: 131.4+ MB
None
```

*Figure 3*. Overview of the Customer Support on Twitter dataset

| | tweet_id | author_id | inbound | created_at | text | response_tweet_id | in_response_to_tweet_id |
|---|---|---|---|---|---|---|---|
| **0** | 1 | sprintcare | False | Tue Oct 31 22:10:47 +0000 2017 | @115712 I understand. I would like to assist y... | 2 | 3.0 |
| **1** | 2 | 115712 | True | Tue Oct 31 22:11:45 +0000 2017 | @sprintcare and how do you propose we do that | NaN | 1.0 |
| **2** | 3 | 115712 | True | Tue Oct 31 22:08:27 +0000 2017 | @sprintcare I have sent several private messag... | 1 | 4.0 |
| **3** | 4 | sprintcare | False | Tue Oct 31 21:54:49 +0000 2017 | @115712 Please send us a Private Message so th... | 3 | 5.0 |
| **4** | 5 | 115712 | True | Tue Oct 31 21:49:35 +0000 2017 | @sprintcare I did. | 4 | 6.0 |
| **2811769** | 2987947 | sprintcare | False | Wed Nov 22 08:43:51 +0000 2017 | @823869 Hey, we'd be happy to look into this f... | NaN | 2987948.0 |
| **2811770** | 2987948 | 823869 | True | Wed Nov 22 08:35:16 +0000 2017 | @115714 wtf!? I've been having really shitty s... | 2987947 | NaN |
| **2811771** | 2812240 | 121673 | True | Thu Nov 23 04:13:07 +0000 2017 | @143549 @sprintcare You have to go to https://... | NaN | 2812239.0 |
| **2811772** | 2987949 | AldiUK | False | Wed Nov 22 08:31:24 +0000 2017 | @823870 Sounds delicious, Sarah! 😋 https://t.c... | NaN | 2987950.0 |
| **2811773** | 2987950 | 823870 | True | Tue Nov 21 22:01:04 +0000 2017 | @AldiUK warm sloe gin mince pies with ice cre... | 2987951,2987949 | NaN |

*Figure 4*. The first and last five rows of the Customer Support on Twitter dataset

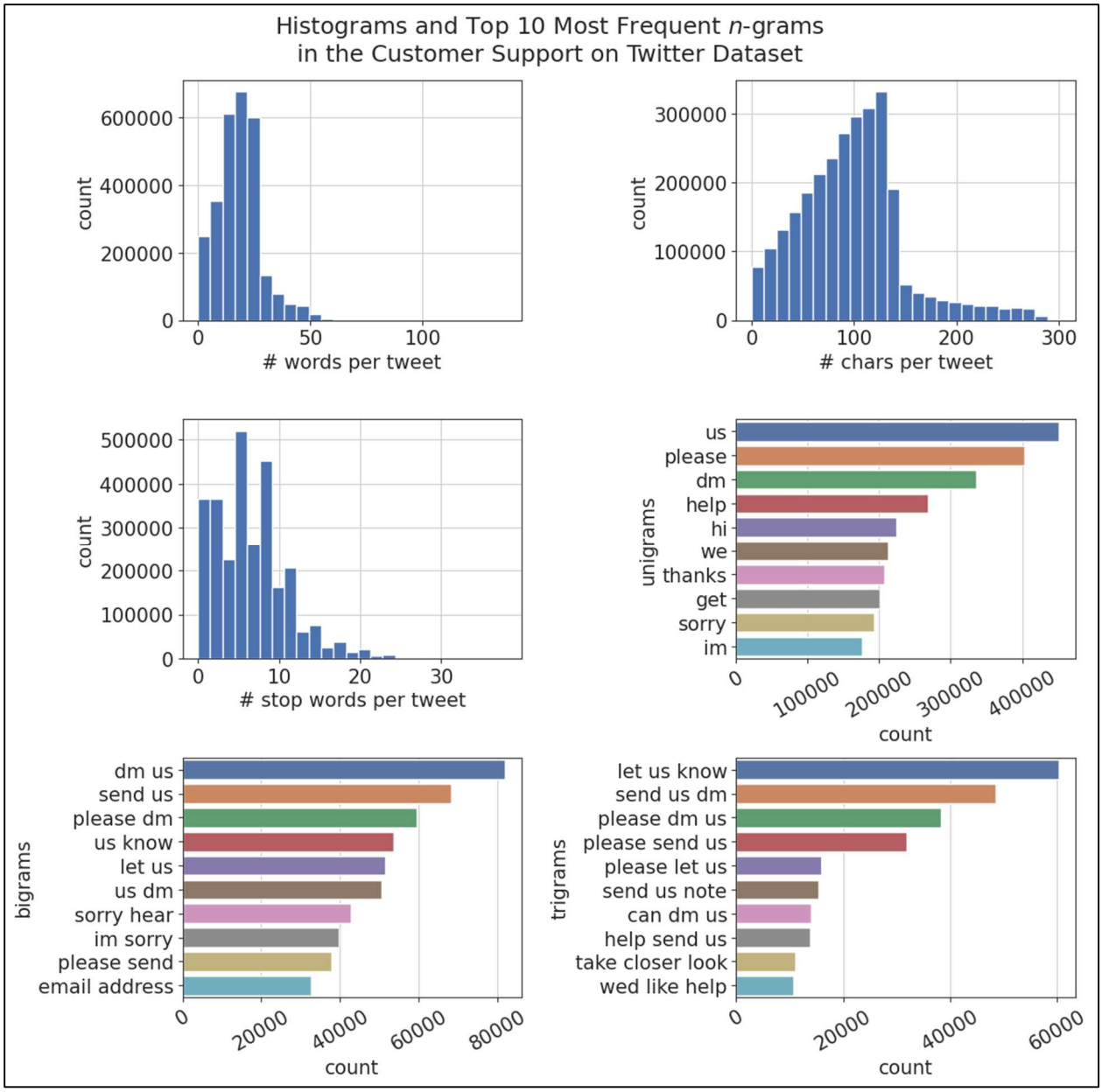


*Figure 5.* Histograms and the top 10 most frequent n-grams in the Customer Support on Twitter dataset

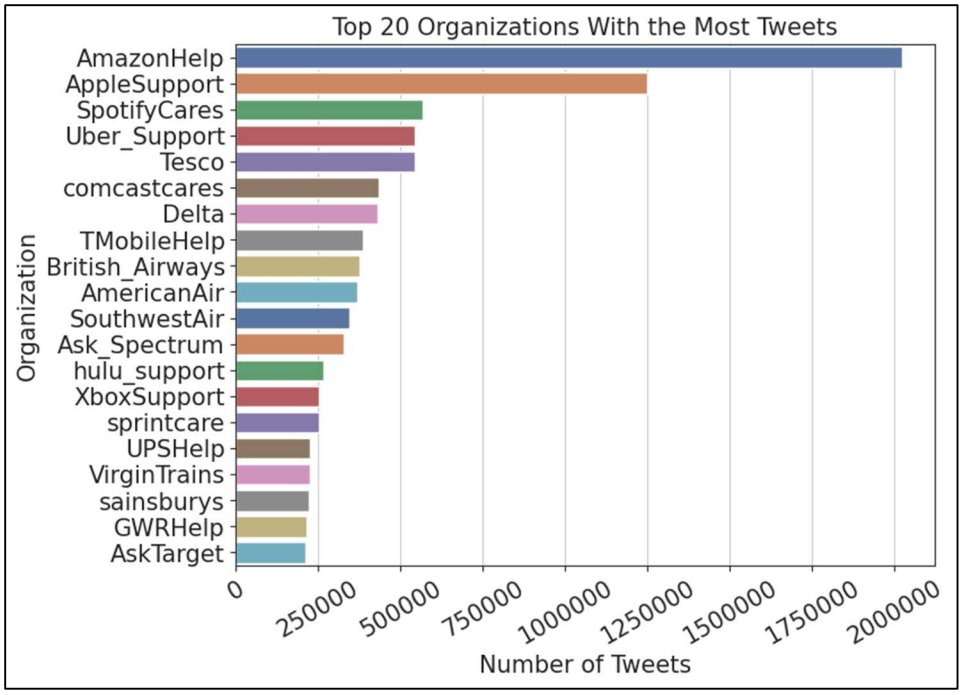


*Figure 6*. Top 20 organizations in the Customer Support on Twitter dataset with the most tweets


*Figure 7*. Word cloud displaying the most frequently occurring words in the Customer Support on Twitter dataset

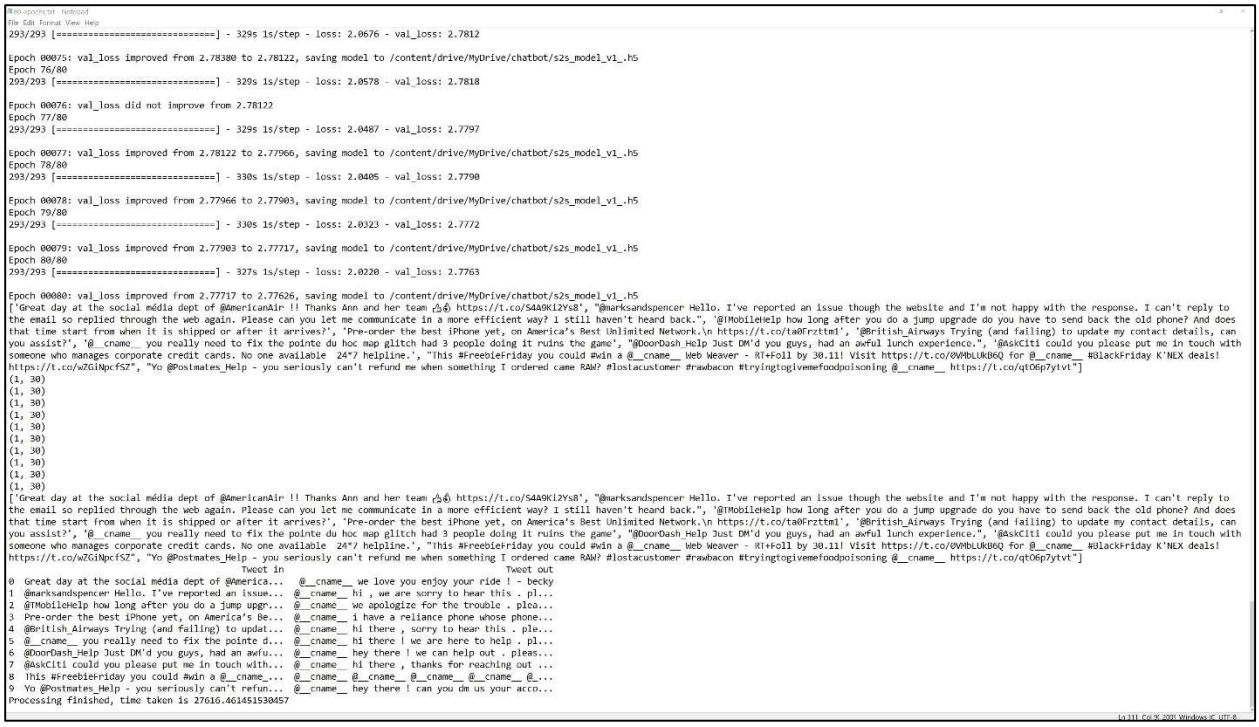


*Figure 8*. The results after training the chatbot for 80 epochs using Google's servers available through Google Colab Pro Notebooks

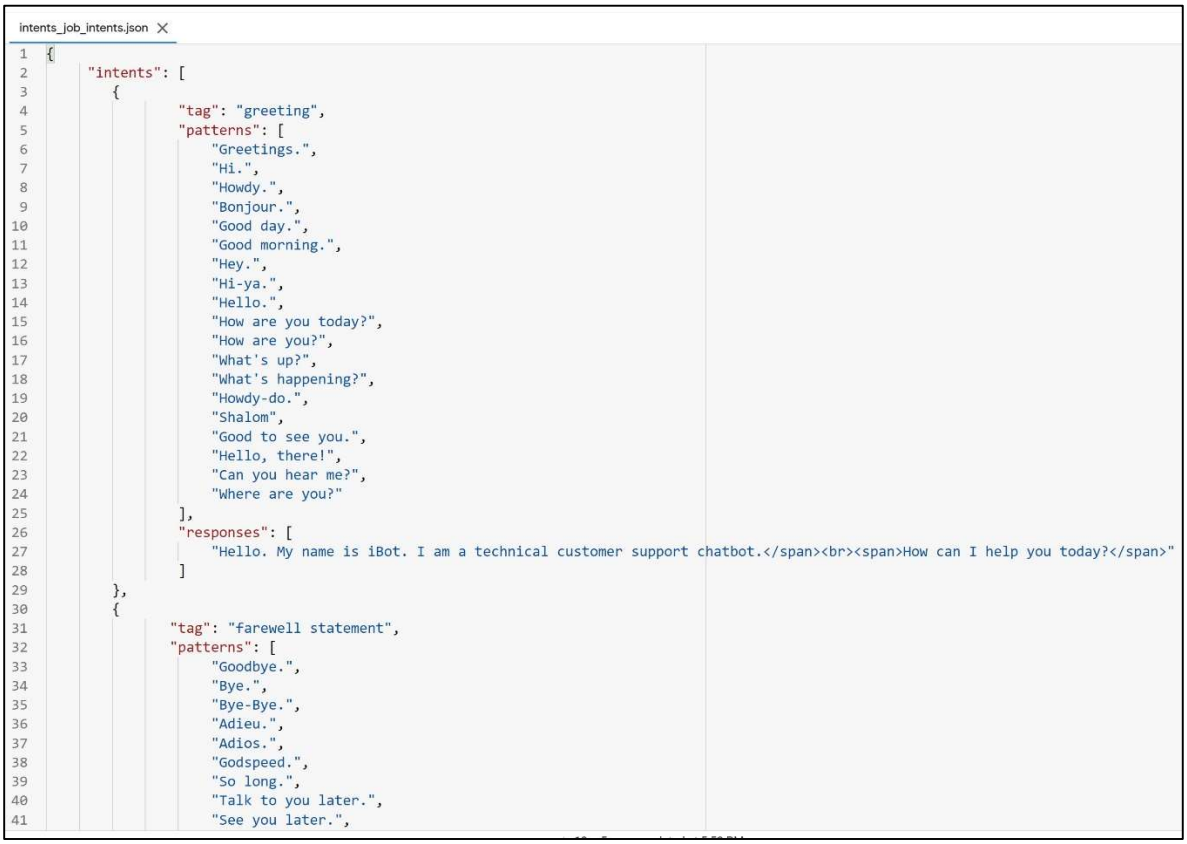


*Figure 9*. A portion of the JSON intents file used to train the chatbot's retrieval-based model

```
49
50  # init training data
51  training = []
52  output_empty = [0] * len(classes)
53  for doc in documents:
54      bag = []
55      pattern_words = doc[0]
56      pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words]
57
58      for w in words:
59          bag.append(1) if w in pattern_words else bag.append(0)
60
61      output_row = list(output_empty)
62      output_row[classes.index(doc[1])] = 1
63
64      training.append([bag, output_row])
65
66  random.shuffle(training)
67  training = np.array(training)
68  # create train and test lists.  X - patterns, y - intents
69  train_x = list(training[:,0])
70  train_y = list(training[:,1])
71  print('Training data created')
72
73  # Create model with 3 layers.  First layer 128 neurons, second layer 64 neurons
74  # and 3rd output layer contains number of neurons equal to number of intents to
75  # predict
76  # output intent with softmax
77  model = Sequential()
78  model.add(Dense(128, input_shape=(len(train_x[0]),), activation='relu'))
79  model.add(Dropout(0.5))
80  model.add(Dense(64, activation='relu'))
81  model.add(Dropout(0.5))
82  model.add(Dense(len(train_y[0]), activation='softmax'))
83
84  # Compile model.  Stochastic gradient descent with Nesterov accelerated
85  # gradient gives good
86  # results for this model
87  sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
88  model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
89
90  # fitting and saving the model
91  hist = model.fit(np.array(train_x), np.array(train_y), epochs=1000, batch_size=5, verbose=1)
92  model.save(os.path.join(intents_path, 'intents_chatbot_model.h5'), hist)
93
94  print('model created')
```

*Figure 10*. Python code that implements training for the chatbot's retrieval-based model

```
[121]
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM, GlobalMaxPooling1D
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy
from keras.callbacks import *
from keras.initializers import Constant

def create_model_pretrained():
    model = Sequential()
    #embedding layer
    model.add(Embedding(size_of_vocabulary,300,
                        input_length=25,
                        embeddings_initializer=Constant(embedding_matrix),
                        trainable=True))
    #lstm layer
    model.add(LSTM(128,return_sequences=True,dropout=0.2))

    #Global Maxpooling
    model.add(GlobalMaxPooling1D())

    #Dense Layer
    model.add(Dense(64,activation='relu'))
    model.add(Dense(len(y_train_all[0]),activation='softmax'))

    #Add loss function, metrics, optimizer
    # Compile model.  Stochastic gradient descent with Nesterov accelerated
    # gradient gives good
    # results for this model
    sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='sparse_categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    #addingcallbacks
    es = EarlyStopping(monitor='val_loss', mode='min', verbose=2, patience=1000)
    mc = ModelCheckpoint(model_path_pretrained, monitor='val_accuracy', mode='max',
                         save_best_only=True, verbose=2)

    print(model.summary())

    return model
```

*Figure 11*. Code to create a model using pre-trained word embeddings

```
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score
from keras.wrappers.scikit_learn import KerasClassifier

pretrained = KerasClassifier(build_fn=create_model_pretrained, epochs=300,
                             batch_size=5, verbose=2)
scratch = KerasClassifier(build_fn=create_model_scratch, epochs=300,
                          batch_size=5, verbose=2)
bag_of_words = KerasClassifier(build_fn=create_model_bag, epochs=300,
                               batch_size=5, verbose=2)
classifiers = {'WordEmbeddings (pre-trained)': pretrained,
               'WordEmbeddings (from scratch)': scratch}

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)

fig, ax = plt.subplots()
for name, model in classifiers.items():
    print(name, model)
    cv_scores = cross_val_score(model,
                                np.array(X_train_all),
                                np.argmax(y_train_all, axis=1),
                                cv=kfold,
                                scoring='accuracy',
                                n_jobs=-1,
                                verbose=2)
    print(cv_scores.mean())
    my_lbl = f'{name} {cv_scores.mean():.3f}'
    ax.plot(cv_scores, '-o', label=my_lbl)

cv_scores = cross_val_score(bag_of_words,
                            np.array(X_train_bag),
                            np.argmax(y_train_bag, axis=1),
                            cv=kfold,
                            scoring='accuracy',
                            n_jobs=-1,
                            verbose=2)

my_lbl = f'BOW {cv_scores.mean():.3f}'
ax.plot(cv_scores, '-o', label=my_lbl)
ax.set_ylim(0.0, 1.1)
ax.set_xlabel('Fold')
ax.set_ylabel('Accuracy')
handles, labels = ax.get_legend_handles_labels()
# sort both labels and handles by accuracy
labels, handles = zip(*sorted(zip(labels, handles), key=lambda t: t[0]))
print(f'label: {labels}, handle: {handles}')

ax.legend(handles, labels, ncol=1, bbox_to_anchor=(1.04,.5),loc='center left')

plt.show()
```

*Figure 12.* Code to generate the cross-validation plots

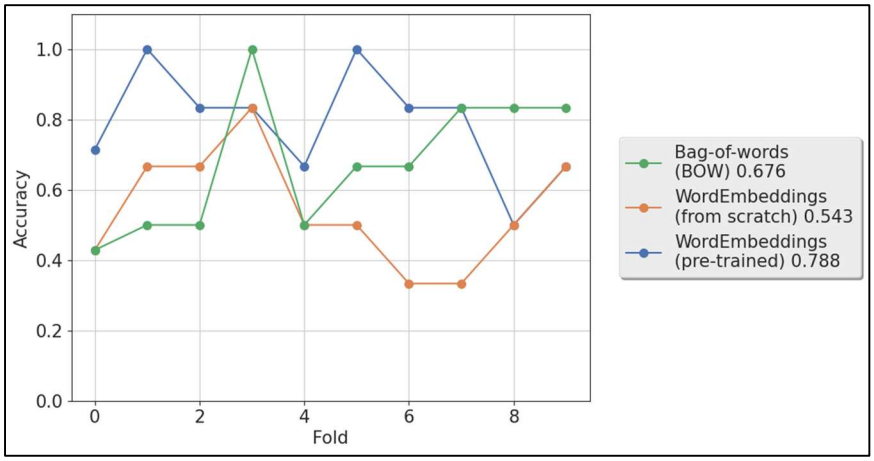


*Figure 13*. 10-fold cross-validation comparing three different word representations

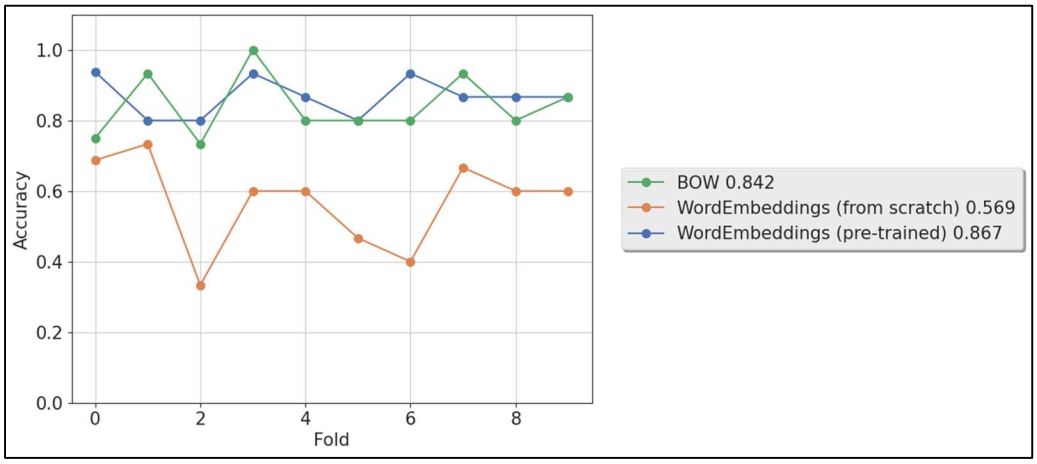


*Figure 14*. 10-fold cross-validation comparing three different word representations

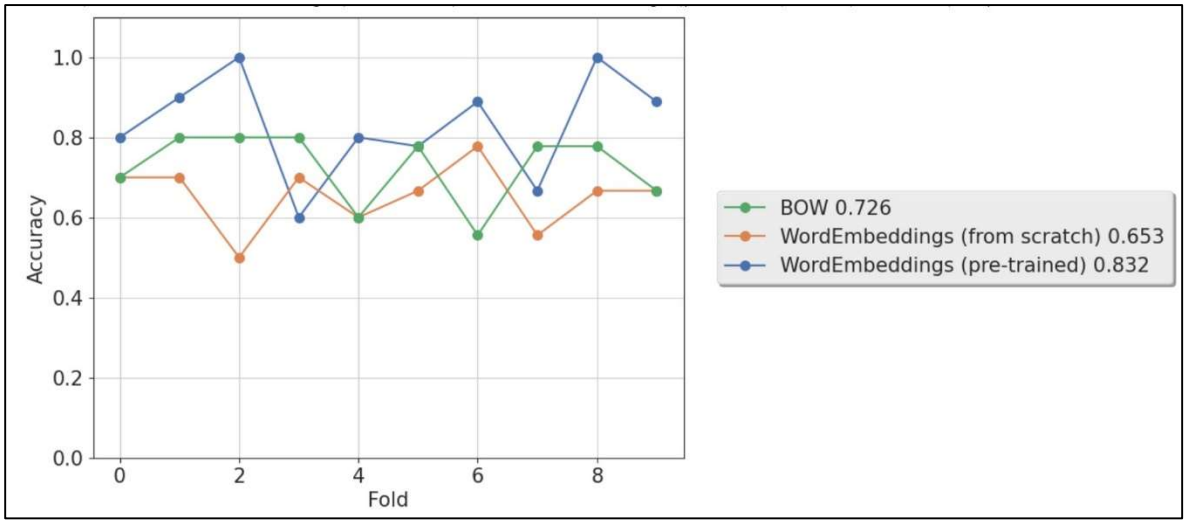


*Figure 15*. 10-fold cross-validation comparing three different word representations

```
        return result

    def string_clean(self, response_orig):

        def upper_repl(match):
            punctuated_inits = \
                '-' + match.group(1).upper() + '.' \
                    + match.group(2).upper() + '.'
            return punctuated_inits

        response = response_orig
        sent_tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
        # remove '@__cname__'
        response = response.replace('@__cname__ ', '')

        # remove spaces before punctuation
        response = re.sub(r'\s([,?.!"](?:\s|$))', r'\1', response)
        # tokenize sentences
        sentences = sent_tokenizer.tokenize(response)
        # captialize senteces
        sentences = [sent.capitalize() for sent in sentences]

        # add html formatting
        sentences = '</span><br><span>'.join(sentences)
        sentences += '</span>'
        # capitalize DM
        sentences = sentences.replace('dm', 'dm'.upper())

        # replace '^' with '-'
        sentences = sentences.replace('^', '-')
        pattern = re.compile(r'- \b([a-z])([a-z])\b')

        sentences = re.sub(pattern, upper_repl, sentences)
        return sentences

    def main(self):
        if self.mode == 'train':
            X_train, X_test, y_train, y_test, test_sentences = self.data_creation()
            print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
            print('Data Creation completed')
            model = self.create_model()
            print('Model creation completed')
            model = self.train_model(model, X_train, X_test, y_train, y_test)
            test_responses = self.generate_response(model, test_sentences)
            print(test_sentences)
            print(test_responses)
            pd.DataFrame(test_responses).to_csv(self.outpath + 'output_response.csv', index=False)

        elif self.mode == 'inference':
            #seq2seq model
            model = load_model(self.load_model_from)
            self.vocabulary = joblib.load(os.path.join(self.outpath, 'vocabulary.pkl'))
            self.reverse_vocabulary = joblib.load(os.path.join(self.outpath, 'reverse_vocabulary.pkl'))
            count_vectorizer = joblib.load(os.path.join(self.outpath, 'count_vectorizer.pkl'))
            self.analyzer = count_vectorizer.build_analyzer()

            #load intent model
            intent_model = load_model(self.intent_load_model_from)
            self.intent_intents = json.loads(open(self.intent_load_intents_from, encoding='cp1252').read())
            self.intent_words = pickle.load(open(self.intent_load_words,'rb'))
            self.intent_classes = pickle.load(open(self.intent_load_classes,'rb'))
            self.tkizer = pickle.load(open(self.t_path,'rb'))

            while True:
                try:
                    userText = request.args.get('msg')
                    ints = self.predict_class(userText, intent_model, method='WE')
                    intent_response = self.getResponse(ints, self.intent_intents)
                    if (intent_response != 'help'):
                        return str(intent_response)
                    elif (intent_response == 'help'):
                        response = self.respond_to_input(model, userText)
                        response = self.string_clean(response)
                        return str(response)

                except(KeyboardInterrupt, EOFError, SystemExit):
                    break


@app.route("/")
def home():
    return render_template("index.html")

@app.route("/get")
def get_bot_response():
    obj = chatbot()
    obj.mode = 'inference'
    response = obj.main()
    return response

app.run()
```

*Figure 16*. Code to generate the chatbot's response


*Figure 17*. Sample conversation had with the chatbot


*Figure 18*. Sample conversation had with the chatbot

## Contents

## OPTION #2: NLP Chatbot Final Version

This paper presents a chatbot that uses Natural Language Processing (NLP) techniques to produce coherent responses to user inputs. NLP is a research area exploring how computers can manipulate speech or text for useful purposes (Chowdhury, 2003). Michael Mauldin, the developer of the first Verbot, Julia, introduced the term chatbot in 1994 (Mondal *et al.*, 2018). Chatbots are text- or voice-based agents that aim to make conversations between humans and machines (Lalwani *et al.*, 2018; Setiaji & Wibowo, 2016).

Chatbots may be open- or closed-domain, as well as retrieval- or generative-based (Britz, 2016). This paper describes a closed-domain chatbot developed in Python that uses a hybridization of generative- and retrieval-based methods to provide technical customer support to users. The paper discusses the internal workings of these methods, including the sequence-to-sequence (Seq2Seq) architecture and neural networks. Further, the paper discusses using 10-fold cross-validation to compare models created with three distinct types of word representations: (a) bag-of-words (BOW), (b) word embeddings from scratch, and (c) Global Vectors for Word Representation (GloVe). Lastly, the paper gives an overview of how the chatbot generates its responses, the Python tools and libraries implemented, and ideas for future research.

### Closed- and Open-Domain Chatbots

Orin (2017) describes chatbots as either open- or closed-domain. Users can take conversations anywhere in open-domain settings, and such conversations typically exist without well-defined intentions or goals (Mondal *et al.*, 2018). Britz (2016) states that because open-domain chatbots require a certain amount of world knowledge to generate responses to the infinite number of topics they can address, they are harder to implement than closed-domain chatbots. Contrarily, the space of inputs and outputs is limited in closed-domain settings, where

chatbots aim to achieve specific goals, such as providing technical customer support or shopping assistance to users. Therefore, closed domain chatbots are easier to implement. This paper describes a closed domain chatbot intended to provide technical customer support to users.

**Retrieval- and Generative-Based Chatbots**

Not only are chatbots closed- or open-domain, but also retrieval- or generative-based. Retrieval-based approaches do not generate new text but use heuristics to choose between predetermined replies, using similarity scores to match between user inputs and intents (Britz, 2016; Setiaji & Wibowo, 2016). On the other hand, Britz (2016) describes generative-based models as harder to implement than retrieval-based approaches. Based on machine translation techniques, generative-based chatbots translate between inputs and outputs rather than between languages, generating new responses from the ground up without relying on handcrafted replies.

Britz (2016) describes both methods as having their advantages and disadvantages. For instance, retrieval-based methods do not make grammatical mistakes but have difficulties handling unseen data. In contrast, generative methods can handle novel data and are, therefore, considered more intelligent than retrieval-based methods. However, they are also more difficult to train, require more training data, and often make grammatical mistakes. This paper describes a chatbot that implements a hybrid approach employing a combination of the two techniques. Researchers use deep learning techniques for both retrieval- and generative-based approaches. On the other hand, Sequence-to-sequence (Seq2Seq) architectures are specific to generative-based models.

**Sequence-to-Sequence Architecture**

Palasundram *et al.* (2019) describe Seq2Seq models as the most researched to implement chatbots and, surprisingly, as still in their infancy. It is useful to talk about Seq2Seq models in

the context of Deep Neural Networks (DNNs). Sutskever *et al.* (2014) define DNNs as powerful machine learning (ML) models with a significant limitation: they require their inputs and outputs to be encoded with vectors of fixed dimensionalities. Therefore, sequential problems, including speech recognition, machine translation, and question answering, best expressed by sequences of varying lengths, pose significant challenges for DNNs. Seq2Seq models address these challenges by using Long Short-Term Memory (LSTM) networks, special types of Recurrent Neural Networks (RNNs). In effect, Seq2Seq models make minimal assumptions about the structure of their input sequences and can learn long-term dependencies between inputs, making them ideal for generative-based chatbots (Britz, 2016; Singh, 2020; Sutskever *et al.,* 2014).

### *Encoder-Decoder Framework*

Palasundram *et al.* (2019) describe Seq2Seq models as based on the Encoder-Decoder framework of RNNs, comprised of three key components: (a) an *embedding layer*, which converts inputs into variable-length vector representations of real numbers; (b) an *encoder*, which produces intermediate states of fixed-length vectors by processing the embedding layer's output; and (c) a *decoder*, which generates a variable-length sentence from the encoder's fixed-length vectors. Further, Dugar (2021) describes a model's encoder as capturing an input sequence's context in a hidden state vector, which it sends to a decoder to produce an output sequence. Hidden state vectors can be initialized to any size, though typical starting values include powers of 2 (i.e., 256, 512, or 1,024). Figure 1 shows that Prakash and Kanagachidambaresan (2021) provide a default setting of 100 for the chatbot's hidden state vector size. Therefore, it may be useful to adjust this hyperparameter in future research to see if it improves the model's performance.

### *Generative-Based Model Overview (Model #1)*

Prakash and Kanagachidambaresan (2021) describe the generative-based chatbot as implementing a Seq2Seq architecture. The architecture assumes the same prior distributions for input and output words. Moreover, the model shares its embedding layer with its encoding and decoding processes, improving its context sensitivity by retaining its encoder's output, also known as the thought vector. The model combines the thought vector with a dense vector representation during response generation to help it remember the input sequence's context, creating an LSTM network in the model's decoder conditioned on its input sequence (Sutskever *et al.*, 2014). Lines 153 – 215 of Figure 2 show the Seq2Seq model implementation in Python code.

**Hyperparameters of the Generative-Based Model (Model #1)**

Palasundram *et al.* (2019) describe one key challenge of Seq2Seq architectures: researchers need to tune many hyperparameters to produce good performing models, including the model's embedding type, embedding size, hidden units' size, and dropout rate. Options for the embedding type include word and character embeddings. Palasundram *et al.* found the former to produce significantly better results than the latter in generative-based models trained on Malay language data in educational settings. Typical embedding sizes range from 100 to 300. Further, Palasundram *et al.* reduced their model's likeliness to overfit its data by fine-tuning its dropout rate. The dropout rate is a regularization technique in neural network-based models that forces some neurons to cover for others by ignoring selected neurons at random, resulting in a network more capable of generalization and less likely to overfit the data (Brownlee, 2016).

Figure 1 shows the Seq2Seq model's initial hyperparameter settings that Prakash and Kanagachidambaresan (2021) provide. The model's initial vocabulary size is set to 50,000 words. Palasundram *et al.* (2019) write that researchers often limit the vocabulary sizes of

Seq2Seq models to prevent them from consuming too many resources and prevent long training times. For instance, it took 7.67 hours to train this chatbot's generative-based model for 80 epochs on 50,000 records of the Customer Support on Twitter dataset using Google's distributed, GPU-accelerated hardware available through Google Colab Pro Notebooks. For comparison, Palasundram *et al.* (2019) trained their generative-based model for 200 epochs. Other noteworthy hyperparameter settings shown in Figure 1 include the size of the model's embedding layer (100), its maximum sequence length (30), and its dropout rate (0.3). An idea for future research is to find the ideal settings for these hyperparameters to balance the model's complexity in terms of its bias-variance tradeoff (Claesen & De Moor, 2015).

### Customer Support on Twitter Dataset

As mentioned, the Seq2Seq model this paper describes was trained for 80 epochs on the Customer Support on Twitter dataset. The dataset is available through Kaggle.com as a 516.53 MB CSV file titled "twcs.csv" (*Customer Support on Twitter | Kaggle*, n.d.). "Customer Support" describes this dataset as a large, modern corpus of mostly English conversations between consumers and customer support agents that offers three main advantages over other conversational text datasets: (a) it is focused, meaning that customers in the dataset discuss a relatively limited number of service-related problems; (b) it is natural, meaning that the dataset samples a broad range of the population using a common form of typed text; and (c) it is succinct, meaning that the dataset offers concise descriptions of problems and solutions due to Twitter's 280-character limit.

#### *Exploratory Data Analysis*

"Customer Support" describes each row in the Customer Support on Twitter dataset as representing a tweet, for which there exists at least one company reply for every customer

request. All sensitive information in the dataset, including customer email addresses, phone numbers, and usernames, have been replaced with masked values. The dataset contains 2,811,774 rows and seven columns: “tweet_id,” “author_id,” “inbound,” “created_at,” “text,” “response_tweet_id,” and “in_response_to_tweet_id.” The “text” column contains each tweet’s text, and the “inbound” column indicates whether a tweet originated from a consumer or an organization.

Figure 3 shows a descriptive overview of the dataset using Python’s *info()* function. Figure 4 shows the dataset’s first five and last five rows. Figure 5 shows histograms displaying the frequency distributions for three dataset characteristics, words per tweet, characters per tweet, and stopwords per tweet, as well as the top 10 most frequently occurring unigrams, bigrams, and trigrams. Stopwords are words that occur frequently in documents but contribute little to their context and are, therefore, meaningless in terms of information retrieval, including words like “the,” “and,” and “or” (Lo *et al.*, 2005). On the other hand, Fenner (2019) describes unigrams, bigrams, and trigrams as single words, adjacent word pairs, and three-word phrases, respectively. For instance, “Let us know” is the most frequently occurring trigram in the data. Next, Figure 6 shows a horizontal bar graph of the dataset’s top 20 most frequently occurring organizations, the top three of which are Amazon, Apple, and Spotify. Finally, Figure 7 displays a word cloud of the most frequently occurring unigrams and bigrams after filtering out words with fewer than four characters. Popular unigrams and bigrams include “customer support,” “happy,” “help,” “please,” and “sorry.”

### Training the Generative-Based Model (Model #1)

Figure 8 shows the results of the chatbot’s initial training. As mentioned, it took 7.67 hours to train the model for 80 epochs using Google’s distributed, GPU-accelerated hardware

available through Google Colab Pro Notebooks. Over 80 training epochs, the model reduced its loss on its validation dataset from 4.78 to 2.78, a 52.91% reduction. The TXT file titled "80-epochs.txt" found in the "training_results*"* directory contains the results from all 80 training epochs, as well as the model's responses to 10 randomly selected input test sentences. Moreover, once the model completes training, it saves its weights and architecture to a Hierarchical Data Format Version 5 (H5) file, a file format commonly used in astronomy, engineering, genomics, and physics, which supports large, complex, and heterogeneous data types (*H5 File Extension*, n.d.; Wasser, 2020). The chatbot then loads this file during its inference stage, which refers to the process of using a trained ML algorithm to make a prediction, or, in this case, generate a response (DeBeasi, 2019).

**Testing the Generative-Based Model (Model #1)**

The generative-based model was combined with the Python micro web framework, *Flask*, to test its performance on novel data using a web browser. The chatbot begins by loading the H5 file saved during training, which it uses to produce responses to novel input. During testing, the chatbot produced relevant results for customers looking to return products but was limited in its scope of replies. For example, the chatbot apologized even when receiving a greeting like "Hello!" Therefore, a second, retrieval-based model was implemented to help the chatbot expand its knowledge base and add more variety to its output.

**Training the Retrieval-Based Model (Model #2)**

Skolo Online (2021) provided the template code for the chatbot's retrieval-based model. Mondal *et al.* (2018) define retrieval-based approaches as using heuristics to match user inputs to files of intents and responding using correlated sets of predetermined responses. Therefore, the retrieval-based model allows the chatbot to predict a user's intent from their input and respond

with a reply linked to that intent. The tasks that the chatbot can complete after adding the retrieval-based approach include greeting customers, bidding them farewell, tracking packages, looking up order numbers, providing customer support, and others.

The chatbot's responses are contained within a JavaScript Object Notation (JSON) file titled "intents_job_intents.json" in the "data/intents" subfolder, a portion of which is shown in Figure 9. The main challenge of this methodology is that developers must handcraft the model's responses. Because all use case scenarios cannot be determined beforehand, the model may produce unexpected results on unseen data. Figure 10 shows the Python code to train the retrieval-based model. The following subsections describe the model in greater detail and present steps to improve its accuracy.

### *Tokenization and Lemmatization*

The model begins by tokenizing the user input found in the JSON file, referred to as patterns, adding these patterns and their corresponding categories to a Python list object titled "documents." The categories represent user intents. The model then lemmatizes these words using the *WordNetLemmatizer* from the *NLTK* package. Lemmatization is the process of reducing a word to its dictionary form (e.g., "jump") given its inflected variances (e.g., "jumped," "jumps," and "jumping") (Bergmanis & Goldwater, 2018). Next, the model converts all words to lowercase, sorts them, and eliminates any duplicates so that only unique words remain. The application then outputs the number of words and classes to the console. Further, it uses the Python *pickle* module to save these objects to disk so it can later access them during inference ("Understanding Python Pickling with Example," 2017).

### *Bag-of-Words Approach*

Next, the model begins training on the processed JSON file. The retrieval-based approach compares several word representation versions, one of which is the bag-of-words (BOW) approach. Fenner (2019) describes the BOW approach as a simple yes/no table of document words. Additionally, Kusner *et al.* (2015) write that the BOW approach is one of the two most common ways to represent documents in NLP applications. They define the methodology as implementing "a vector of word counts of dimensionality *d*, the size of the [model's] dictionary" (Kusner *et al.*, 2015, p. 961). Further, Martins and Matsubara (2003) describe the BOW approach as representing text documents in their tabular form, with each row representing a document and each column a word.

### *Neural Network Architecture*

After creating word representations for the dataset's feature and target variables, the program adds them to a Python list object as conjoined tuples. The application then shuffles this preprocessed training data, converting it into an *n*-dimensional *NumPy* array before splitting it into feature and target representations to feed into its neural network. The retrieval-based model of the chatbot uses the *Sequential* class of the *Keras* API to implement a neural network for the BOW approach. Skolo Online (2021) states that the first layer of this network contains 128 neurons, the second layer 64 neurons, and the output layer the number of user intents to predict.

#### Softmax Function and Stochastic Gradient Descent

Skolo Online (2021) states that the output layer of the neural network uses the softmax function as its activation function. The softmax function converts the network's weighted sum values into probabilities representing the likelihood of an input belonging to a particular class (Brownlee, 2020). Skolo Online states that using a Stochastic Gradient Descent (SGD) with Nesterov momentum performs well for this model. Nesterov momentum is a general approach to

modifying gradient descent-type methods to improve their initial convergence (*Nesterov's Gradient Acceleration - Calculus*, n.d.).

### *Cross-Validation*

The chatbot explores the effects of multiple word representations on its retrieval-based approach using 10-fold cross-validation. Brownlee (2018) describes cross-validation as a resampling method to evaluate ML models on limited and unseen data. Notably, in cross-validation, each record is used in the holdout set 1 time and used to train the model $k - 1$ times. This study compares three types of word representations in its retrieval-based approach: (a) Bag-of-words (BOW), (b) word embeddings created from scratch, and (c) pre-trained word embeddings obtained from a Global Vectors for Word Representation (GloVe) file.

### *Word Embeddings, Global Vectors for Word Representation, and Transfer Learning*

Levy and Goldberg (2014) define word embeddings as dense vector representations of words derived from neural network-based training methods that convey their semantic and syntactic similarities. Additionally, Pennington *et al.* (2014) describe GloVe as an unsupervised learning algorithm to obtain word vector representations. The GloVe file, titled "glove.840B.300d.txt," is 2.03 GB, contains 840 billion tokens, and has a vocabulary of size 2.2 million. The application downloads the file from its corresponding URL if it does not already exist. The file is so large that it requires using Google's servers to train the chatbot.

Moreover, using the pre-trained word embeddings available from the GloVe file demonstrates an example of transfer learning (TL). Brownlee (2017) describes TL as reusing models developed for one purpose as the starting point for others. In the case of GloVe, training is performed on global word-word co-occurrence counts, producing a word vector space with a

meaningful substructure that achieved 75% accuracy on the word analogy dataset (Pennington *et al.*, 2014).

Figures 11 – 12 present the Python code to construct the pre-trained embedding model and the line graph that compares representations across three 10-fold cross-validations. Figures 13 – 15 display these graphs. The pre-trained word embeddings (*avg. acc.* = 82.9%) outperformed the other two approaches on all three occasions. Additionally, the BOW approach (*avg. acc.* = 74.8%) consistently outperformed the embeddings created from scratch (*avg. acc.* = 58.8%). Therefore, both the BOW model and the model using pre-trained embeddings are trained on the entire dataset and saved for the chatbot to use during inference. Inference refers to putting a model into action on live data to produce actionable output and is what the chatbot uses to generate its responses (*What Is Machine Learning Inference?*, n.d.).

**Generating Responses via Model Hybridization**

To generate responses during inference, the architecture loads both its generative- and retrieval-based models. When users converse with the chatbot, it retrieves the user's input from the chat textbox and uses its retrieval-based model to predict a user's intent. If the model predicts the user needs help with a technical problem, such as returning a product, the chatbot uses its generative-based model to create a response from scratch. If, however, the chatbot predicts the user is not requesting help for a technical problem but has a different intent, it responds using its retrieval-based approach to select a response from a repository of handcrafted replies. Therefore, the chatbot can respond to a wider variety of user inputs by hybridizing its generative- and retrieval-based approaches.

Figure 16 shows the Python code the model uses to generate responses. Figures 17 – 18 show two sample conversations had with the chatbot. To run the chatbot, all one needs to do is

install the necessary libraries, described next, and run the "app.py" file. However, the chatbot performs better using Google's distributed, GPU-accelerated hardware than when run locally. Additionally, each time the chatbot's JSON file is updated, the "intents_training.py" file needs to be re-run to re-train the chatbot, which also requires using Google's distributed, GPU-accelerated servers.

For these reasons, this submission also includes the .ipynb and .py files corresponding to the Google Colab Notebook files located in its "colab" folder. To train the generative-based model requires updating the chatbot's mode to "training" rather than "inference." Finally, a GitHub repository at the following URL mirrors the files included with this submission: https://github.com/sminerport/CSC525-Principles-of-ML-Chatbot. However, GitHub does not accept files greater than 100MB, excluding the H5 file of the Seq2Seq model and the Customer Support on Twitter dataset. Additional instructions for training and running the chatbot are available in the "README.md" file.

***Python Tools and Libraries***

The following gives a brief overview of the Python tools and libraries implemented by the chatbot, including *NumPy*, *pandas*, *scikit-learn*, *TensorFlow, Keras, NLTK,* and *Flask*. *NumPy* is a Python library for scientific computing, at the core of which is the *ndarray*, a multidimensional array object. *NumPy* provides various routines, including mathematical and logical, for fast array operations (*What Is NumPy? — NumPy v1.21 Manual*, n.d.). *pandas* is a Python package that provides expressive data structures for working with relational data, including Series and DataFrames (*Package Overview — Pandas 1.3.3 Documentation*, n.d.). *scikit-learn* is a simple and efficient open-source tool for predictive analytics, which provides many supervised and unsupervised ML algorithms (*What Is Scikit-Learn?*, n.d.). *TensorFlow* is

an end-to-end open-source platform for ML with a comprehensive ecosystem of tools that allow researchers to push the state-of-the-art in ML (*TensorFlow*, n.d.).

Additionally, *Keras* is an industry-strength framework built on top of *TensorFlow* that covers every step of the ML workflow and can scale to large clusters of GPUs (*Keras: The Python Deep Learning API*, n.d.). *NLTK* stands for the "Natural Language Toolkit," a platform that allows Python programs to handle human language data by providing a suite of text processing libraries for classification, tokenization, stemming, tagging, and parsing (*Natural Language Toolkit — NLTK 3.6.3 Documentation*, n.d.). Finally, *Flask* is a Python API for developing web applications ("Python | Introduction to Web Development Using Flask," 2018). Implementing the *Flask* framework allows for conversing with the chatbot via a web browser.

## Conclusion

In conclusion, this paper described a closed-domain chatbot that used a hybridization of generative- and retrieval-based methods to provide technical customer support to users. The paper described open- and closed-domain chatbots, as well as those that are retrieval- and generative-based. The paper also discussed sequence-to-sequence (Seq2Seq) architectures, encoder and decoder frameworks, model hyperparameters, the Customer Support on Twitter dataset, and retrieval-based approaches like tokenization and lemmatization, bag-of-words (BOW), and neural networks. By hybridizing its generative- and retrieval-based approaches, the chatbot increased its depth of knowledge and variety of responses. Further, 10-fold cross-validation was used to compare the effect that different word representations had on the chatbot's retrieval-based model, the most effective of which was found to be the pre-trained word embeddings provided by GloVe. Finally, the paper discussed how the chatbot generates its responses, the Python tools and libraries used to construct it, and ideas for future research.

References

Bergmanis, T., & Goldwater, S. (2018). Context-Sensitive Neural Lemmatization with Lematus. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 1391–1400. https://doi.org/10.18653/v1/N18-1126

Britz, D. (2016, April). Deep Learning for Chatbots, Part 1 – Introduction. *KDnuggets*. https://www.kdnuggets.com/deep-learning-for-chatbots-part-1-introduction.html/

Brownlee, J. (2016, June 19). Dropout Regularization in Deep Learning Models with Keras. *Machine Learning Mastery*. https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

Brownlee, J. (2017, December 19). A Gentle Introduction to Transfer Learning for Deep Learning. *Machine Learning Mastery*. https://machinelearningmastery.com/transfer-learning-for-deep-learning/

Brownlee, J. (2018, May 22). A Gentle Introduction to k-fold Cross-Validation. *Machine Learning Mastery*. https://machinelearningmastery.com/k-fold-cross-validation/

Brownlee, J. (2020, October 18). Softmax Activation Function with Python. *Machine Learning Mastery*. https://machinelearningmastery.com/softmax-activation-function-with-python/

Chowdhury, G. G. (2003). Natural language processing. *Annual Review of Information Science and Technology*, *37*(1), 51–89.

Claesen, M., & De Moor, B. (2015). Hyperparameter Search in Machine Learning. *ArXiv:1502.02127 [Cs, Stat]*. http://arxiv.org/abs/1502.02127

*Customer Support on Twitter | Kaggle*. (n.d.). Retrieved September 26, 2021, from https://www.kaggle.com/thoughtvector/customer-support-on-twitter

DeBeasi, P. (2019, February 14). Training versus Inference. *Paul DeBeasi*. https://blogs.gartner.com/paul-debeasi/2019/02/14/training-versus-inference/

Fenner, M. (2019). *Machine Learning in Python for Everyone*. Addison-Wesley.

*H5 File Extension—What is an .h5 file, and how do I open it?* (n.d.). Retrieved October 7, 2021, from https://fileinfo.com/extension/h5

*Keras: The Python deep learning API*. (n.d.). Retrieved October 5, 2021, from https://keras.io/

Kusner, M., Sun, Y., Kolkin, N., & Weinberger, K. (2015). From word embeddings to document distances. *International Conference on Machine Learning*, 957–966.

Lalwani, T., Bhalotia, S., Pal, A., Bisen, S., & Rathod, V. (2018). Implementation of a Chatbot System using AI and NLP. *International Journal of Innovative Research in Computer Science & Technology*, *6*, 26–30. https://doi.org/10.21276/ijircst.2018.6.3.2

Levy, O., & Goldberg, Y. (2014). Dependency-Based Word Embeddings. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 302–308. https://doi.org/10.3115/v1/P14-2050

Lo, R. T.-W., He, B., & Ounis, I. (2005). Automatically building a stopword list for an information retrieval system. *Journal on Digital Information Management: Special Issue on the 5th Dutch-Belgian Information Retrieval Workshop (DIR)*, *5*, 17–24.

Martins, C., & Matsubara, E. (2003). *Reducing the dimensionality of bag-of-words text representation used by learning algorithms*.

Mondal, A., Dey, M., Das, D., Nagpal, S., & Garda, K. (2018). Chatbot: An automated conversation system for the educational domain. *2018 International Joint Symposium on Artificial Intelligence and Natural Language Processing (ISAI-NLP)*, 1–5. https://doi.org/10.1109/iSAI-NLP.2018.8692927

*Natural Language Toolkit—NLTK 3.6.3 documentation*. (n.d.). Retrieved October 5, 2021, from https://www.nltk.org/

*Nesterov's gradient acceleration—Calculus*. (n.d.). Retrieved October 5, 2021, from https://calculus.subwiki.org/wiki/Nesterov%27s_gradient_acceleration

Orin, T. D. (2017). *Implementation of a Bangla chatbot*. BRAC University.

*Package overview—Pandas 1.3.3 documentation*. (n.d.). Retrieved October 5, 2021, from https://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543.

Prakash, K. B., & Kanagachidambaresan, G. R. (2021). *Programming with TensorFlow: Solution for Edge Computing Applications*. Springer Nature.

Python | Introduction to Web development using Flask. (2018, October 16). *GeeksforGeeks*. https://www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/

Setiaji, B., & Wibowo, F. W. (2016). Chatbot Using a Knowledge in Database: Human-to-Machine Conversation Modeling. *2016 7th International Conference on Intelligent Systems, Modelling, and Simulation (ISMS)*. https://doi.org/10.1109/ISMS.2016.53

Singh, P. (2020, January 14). *LSTM- Long Short-Term Memory*. Medium. https://medium.com/analytics-vidhya/lstm-long-short-term-memory-5ac02af47606

Skolo Online. (2021, March 27). *Create a 🤖Deep Learning 🤖 Machine Learning Chatbot with Python and Flask*. https://www.youtube.com/watch?v=8HifpykuTI4

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *ArXiv:1409.3215 [Cs]*. http://arxiv.org/abs/1409.3215

*TensorFlow*. (n.d.). TensorFlow. Retrieved October 5, 2021, from https://www.tensorflow.org/

Understanding Python Pickling with example. (2017, June 8). *GeeksforGeeks*. https://www.geeksforgeeks.org/understanding-python-pickling-example/

Wasser, L. (2020, October 7). *Hierarchical Data Formats—What is HDF5? | NSF NEON | Open Data to Understand our Ecosystems*. https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5

*What is Machine Learning Inference?* (n.d.). Hazelcast. Retrieved October 9, 2021, from https://hazelcast.com/glossary/machine-learning-inference/

*What is NumPy? —NumPy v1.21 Manual*. (n.d.). Retrieved October 5, 2021, from https://numpy.org/doc/stable/user/whatisnumpy.html

*What is Scikit-Learn?* (n.d.). Codecademy. Retrieved October 5, 2021, from https://www.codecademy.com/articles/scikit-learn