Demo Steps: Deep Learning Sentiment Prediction using the Python API

The data set **CFPB_COMPLAINTS_CLEAN** contains the cleaned text from the original **CFPB_COMPLAINTS** data set. For the text in each complaint, stop words and non-letters have been removed, words have been stemmed, and all tokens were changed to lower case.

Name	Model Role	Measurement Level	Description
DISPUTE	Target	Binary	1 = consumer disputed company response 0 = did not dispute
COMPLAINT	Input	Text	cleaned consumer-submitted complaint

The data set **CFPB_COMPLAINTS_EMBED** contains the Global Vectors for Word Representation (GLOVE) for each term in the **CFPB_COMPLAINTS_CLEAN** data set. The GLOVE was created from word-word co-occurrence statistics from the **CFPB_COMPLAINTS_CLEAN** corpus using an unsupervised learning algorithm. The vectors of dimension 100 show the linear substructure of the word vector space.

Name	Model Role	Measurement Level	Description
VOCAB.TERM	Input	Nominal	Individual terms from the cleaned corpus
X1-X100	Input	Interval	Word representations in 100 dimensions

Note, this demonstration is part II of the text mining notebook and assumes you have run part I. That is, you are working in the same CAS session for the entire notebook.

1. Load the **cfpb_complaints_clean.csv** file onto the CAS server and print the dimension and head of the table.

```
castbl = conn.read_csv(
   os.environ.get("HOME")+"/Courses/EVMLOPRC/DATA/cfpb_complaints_clean.csv",
   casout = dict(name="cfpb_clean", replace=True))

indata = 'cfpb_clean'

display(castbl.shape)
castbl.head()
```

   Notice that stop words and non-letters have been removed, words have been stemmed, and all tokens were changed to lower case. Although it's more challenging to read, the important features of the paragraphs remain.

2. Use the freq action from the simple action set to find the frequency of each dispute level.

```
conn.loadActionSet('simple')
actions = conn.builtins.help(actionSet='simple')

conn.simple.freq(
    table = indata,
    inputs = "dispute"
)
```

   Approximately 50% of the claims were disputed in this sample data set.

3. Partition the data into 80% training, 10% validation and 10% test using the srs action from the sampling action set. Then use the freq action to find the frequency of observations in each set.

```
conn.loadActionSet('sampling')
actions = conn.builtins.help(actionSet='sampling')

conn.sampling.srs(
    table   = indata,
    samppct = 80,
    samppct2 = 10,
    seed = 802,
    partind = True,
    output  = dict(casOut = dict(name = indata, replace = True),
      copyVars = 'ALL')
)
```

   Selected arguments:

Argument	Description
samppct	specifies the sample percentage to be used in sampling or partitioning.
samppct2	specifies a second sample percentage to be used to partition data.

Note: the 80% training is associated with a value of 1 in the partition indicator, the 10% validation is specified by a value of 2, and the testing reverts back to a value of 0.

```
conn.simple.freq(
    table = indata,
    inputs = '_PartInd_'
)
```

4. Use the shuffle action from the table action set to prevent any ordering bias in the data when its fed into the deep learning model.

```
conn.table.shuffle(
    table = indata,
    casOut = dict(name=indata, replace=True)
)
```

5. Load the **cpfb_complaints_embed.csv** file, name it **glove** as a CAS table, and print the dimension and first few observations.

```
embed = conn.read_csv(
   os.environ.get("HOME")+"/Courses/EVMLOPRC/DATA/cfpb_complaints_embed.csv",
   casout = dict(name="glove", replace=True))

display(embed.shape)
embed.head()
```

The word embedding data has 15,930 terms and 100 dimensions. Both of these can be tuned inside the GLOVE algorithm. More terms and larger dimensions take longer training times for the unsupervised GLOVE algorithm.

6. Find the word representations using a DATA step for the terms "credit", "tax", "loan", "debt", "default", "unfair", "difficult", "conflict", "fight", and "harm" and save the information in a new table called **embed_sample**.

```
conn.dataStep.runCode(code=
    '''
    data embed_sample;
        set glove;
        if vocab.term in ('credit','tax','loan','debt','default',

          'unfair','difficult','conflict','fight','harm');
    run;
    '''
)
```

7. Download the **embed_sample** table to the client and plot the first two dimensions of the ten terms to view their vector representations.

```
embed_sample = conn.CASTable(name='embed_sample')
embed10 = embed_sample.to_frame()
display(embed10[['vocab.term','X1','X2']])

plt.figure(figsize=(8,8))
plt.scatter(embed10['X1'], embed10['X2'])
plt.title('Word Representation', fontsize=20)
plt.xlabel('Dimension 1', fontsize=15)
plt.ylabel('Dimension 2', fontsize=15)
plt.xlim(-1.2,1.2)
plt.ylim(-1.2,1.2)
for i in range(10):
    plt.text(embed10['X1'][i]+.03,embed10['X2'][i]+.03,
    embed10['vocab.term'][i], fontsize=15)
plt.show()
```

The terms *default* and *unfair* are the closest to each other, meaning consumers tend to use the term *unfair* when describing a default experience in their complaint. Likewise, the terms *difficult* and *loan* are also close to one another in these two dimensions, providing a similar interpretation.

On the other hand, the terms *debt* and *conflict* are far apart. The term *debt* is in the first quadrant, and *conflict* is in the third quadrant of the plot, meaning these terms rarely co-occur. We could create another graphic with the same words but using another dimension to see how they cluster in a different dimension, or we can plot a different set of terms to interpret their vector representations.

8. Load the deepLearn action set and then build a Recurrent Neural Network with two layers and 30 neurons in each layer.

```
conn.loadActionSet('deepLearn')
actions = conn.builtins.help(actionSet='deepLearn')

conn.deepLearn.buildModel(
    model = dict(name='rnn',replace=True),
    type = 'RNN'
)

conn.deepLearn.addLayer(
    model = 'rnn',
    layer = dict(type='input'),
    replace=True,
    name = 'data'
)
```

```
conn.deepLearn.addLayer(
    model = 'rnn',
    layer = dict(type='recurrent', n=30, act='sigmoid',
       init='xavier', rnnType='rnn', outputType='samelength'),
    srcLayers = 'data',
    replace=True,
    name = 'rnn1'
)

conn.deepLearn.addLayer(
    model = 'rnn',
    layer = dict(type='recurrent', n=30, act='sigmoid',
       init='xavier', rnnType='rnn', outputType='encoding'),
    srcLayers = 'rnn1',
    replace=True,
    name = 'rnn2'
)

conn.deepLearn.addLayer(
    model = 'rnn',
    layer = dict(type='output', act='auto', init='xavier',
       error='auto'),
    srcLayers = 'rnn2',
    replace=True,
    name = 'output'
)

conn.deepLearn.modelInfo(
    model='rnn'
)
```

In order to build a deep learning model, you initialize the model with the buildModel action and then iteratively add layers to the model with the addLayer action. By default, you need an input and output layer. The other layers define the type and complexity of your deep learning neural network. You can view the build model with the modelInfo action to ensure it is built as intended.

Selected arguments for the buildModel action:

Argument	Description
model	specifies an in-memory table that is used to store the model.
type	specifies the model type. ◦ CNN creates an empty model for building a convolutional neural network. ◦ DNN creates an empty model for building a deep, fully connected neural network. ◦ RNN creates an empty model for building a recurrent neural network.

Selected arguments for the addLayer action:

Argument	Description
model	specifies the in-memory table that is the model.
name	specifies a unique name for the layer. As you add layers to the model, you specify this name in the srcLayers parameter. The name is case insensitive.
srcLayers	specifies the names of the source layers for this layer.
layer	specifies the layer type and related parameters for the layer. The value that you specify for type determines the other parameters that apply.

Selected layer argument options:

Argument	Description
type	can be set to BATCHNORM, CONVO, FC, INPUT, OUTPUT, POOL, PROJECTION, RESIDUAL, and RECURRENT.
n	specifies the number of neurons for the layer.
act	specifies the activation function for the layer. The default AUTO specifies to assign the activation function automatically. For a convolutional layer, RECTIFIER is used. For a fully connected layer, TANH is used. For a pooling layer, IDENTITY is used. For an output layer, classification models use SOFTMAX and other models use IDENTITY. For a recurrent layer, LSTM and GRU types use IDENTITY and RNN uses TANH.
init	specifies the initialization scheme for the neurons.
rnnType	specifies the type for the RNN layer. ◦ GRU specifies the layer type is a gated recurrent unit. ◦ LSTM specifies the layer type is a long short-term memory unit.

	◦ RNN specifies the layer type is a recurrent neural network.
outputType	specifies the output type for the RNN layer. ◦ ARBITRARYLENGTH specifies to generate a sequence with arbitrary length. ◦ ENCODING specifies to generate a fixed-length vector. ◦ SAMELENGTH specifies to generate a sequence with the same length as the input.
error	specifies the error function. This function is also known as the loss function. Auto specifies to set the error function according to the model. For a classification model, the error function is ENTROPY. For all other models, the error function is NORMAL. The error can be set to AUTO, CTC, ENTROPY, FCMPERR, GAMMA, NORMAL, and POISSON.

The outputType for the first hidden layer is set to samelength, meaning this layer will generate a sequence with the same length as the input. That is, the sequence of inputs is converted into a sequence of hidden layer values. The next hidden layer output type is encoding. The encoding option can be thought of as a many-to-one transformation, in that we are taking the sequence and converting it to a single value in order to predict the output.

The output type depends on the problem at hand and how the inputs are used to model the RNN output. For example, we could use a many-to-many mapping to model language translation because the number of words needed to speak a phrase in one language might require a different number of words in another language. In this case, we are converting our sequence into a binary prediction.

9. Use the dlTrain action to train the RNN. Use the Adam optimization method and 20 epochs to optimize the weights.

```
conn.deepLearn.dlTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    validTable = dict(name = indata, where = '_PartInd_ = 2'),
    target = 'dispute',
    inputs = 'complaint',
    texts = 'complaint',
    textParms = dict(initInputEmbeddings=dict(name='glove')),
    nominals = 'dispute',
    seed = '649',
    modelTable = 'rnn',
    modelWeights = dict(name='rnn_trained_weights', replace=True),
    optimizer = dict(miniBatchSize=100, maxEpochs=20,
           algorithm=dict(method='adam', beta1=0.9, beta2=0.999,
           learningRate=0.001, clipGradMax=100, clipGradMin=-100))
)
```

Selected arguments:

Argument	Description
table	specifies the settings for an input table.
validTable	specifies the table with the validation data. The validation table must have the same columns and data types as the training table.
text	specifies the character variables to treat as raw text. These variables must be specified in the inputs parameter.
textParms	specifies text parameter options.
initInputEmbeddings	specifies the pretrained word embedding data table. The first variable must be the trained terms and the other columns are the word representations.
modelTable	specifies the in-memory table that is the model.
modelWeights	specifies an in-memory table that is used to store the model weights.
optimizer	specifies the settings for the optimization algorithm, optimization mode, and other settings such as a seed, the maximum number of epochs, and so on.

Selected optimizer argument options:

Argument	Description
algorithm	specifies the algorithm used to find the model weights. The available methods are ADAM, LBFGS, MOMENTUM, and VANILLA. The value that you specify for method determines the other parameters that apply.
miniBatchSize	specifies the number of observations per thread in a mini-batch. You can use this parameter to control the number of observations that the action uses on each worker for each thread to compute the gradient prior to updating the weights. Larger values use more memory. When synchronous SGD is used (the default), the total mini-batch size is equal to miniBatchSize * number of threads * number of workers. When asynchronous SGD is used (by specifying the elasticSyncFreq parameter), each worker trains its own local model. In this case, the total mini-batch size for each worker is miniBatchSize * number of threads.
maxEpochs	specifies the maximum number of epochs. For SGD with a single-machine server or a session that uses one worker on a distributed server, one epoch is reached when the action passes through the data one time. For a session that uses more than one worker, one epoch is reached when all the workers exchange the weights with

	the controller one time. The syncFreq parameter specifies the number of times each worker passes through the data before exchanging weights with the controller.
beta1	specifies the exponential decay rate for the first moment in the Adam learning algorithm.
beta2	specifies the exponential decay rate for the second moment in the Adam learning algorithm.
gamma	specifies the gamma for the learning rate policy.
learningRate	specifies the learning for stochastic gradient descent.
clipGradMax	specifies the maximum gradient value. All gradients that are greater than the specified value are set to the specified value.
clipGradMin	specifies the minimum gradient value. All gradients that are less than the specified value are set to the specified value.
stepSize	specifies the step size when the learning rate policy is set to STEP.
lrPolicy	specifies the learning rate policy. ◦ FIXED specifies a fixed learning rate. ◦ IIN specifies to set the learning rate after each epoch according to the initial learning rate, the value of the gamma parameter, and the value of the power parameter. The rate is calculated as the learningRate* (1+gamma*current epoch)^(-power). ◦ MULTISTEP specifies to set the learning rate after each of the epochs that are specified in the steps parameter. The learning rate is multiplied by the gamma parameter value. ◦ POLY specifies to set the learning rate after each epoch according to the initial learning rate, the maximum number of epochs, and the value of the power parameter. The rate is calculated as the learningRate*(1-current epoch / maxEpochs)^power. ◦ STEP specifies to set the learning rate to the current learning rate multiplied by the gamma parameter value. The number of steps is specified in the stepSize parameter. The rate is recalculated for each group of epochs, according to the step size.

Adam optimization is an extension of stochastic gradient descent. The Adam method applies adjustments to the step size for each individual model parameter in an adaptive manner by approximating second-order information about the objective function based on the previous minibatch gradients. The "adaptive" nature of the algorithm's weight movements is where the name Adam comes from.

We're specifying a small learning rate to inch slowly toward convergence, but the Adam algorithm does adapt these movements based on our specified hyperparameters. The use of Adam also requires us to specify the parameters beta1 and beta2, which are the exponential decay rates for the first and second moment estimates of the adaptive approximation.

Because these rates decay over each epoch, it's a best practice to use values close to 1 so that they stay active for longer periods. After many epochs, the beta parameters converge to zero. As a result, the Adam algorithm no longer corrects the step size for updating the weights.

10. Use the dlScore action to score the RNN on the test data set.

```
conn.deepLearn.dlScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = 'rnn',
    initWeights = 'rnn_trained_weights',
    copyVars = 'dispute',
    textParms = dict(initInputEmbeddings=dict(name='glove')),
    casout = dict(name='rnn_scored', replace=True)
)
```

Selected arguments:

Argument	Description
initWeights	specifies an in-memory table that contains the model weights. These weights are used to initialize the model.
copyVars	specifies the variables to transfer from the input table to the output table.
textParms	species the word embedding table used in the dlTrain action.

11. Build a Gated Recurrent Unit Neural Network to add complexity by using gates to regulate the flow of information. To use a similar number of model weights, use two hidden layers with only 15 neurons in each.

```
conn.deepLearn.buildModel(
    model = dict(name='gru', replace=True),
    type = 'RNN'
)

conn.deepLearn.addLayer(
    model = 'gru',
    layer = dict(type='input'),
    replace=True,
    name = 'data'
)

conn.deepLearn.addLayer(
```

```
    model = 'gru',
    layer = dict(type='recurrent', n=15, act='auto', init='xavier',
                 rnnType='gru', outputType='samelength'),
    srcLayers = 'data',
    replace=True,
    name = 'rnn1'
)

conn.deepLearn.addLayer(
    model = 'gru',
    layer = dict(type='recurrent', n=15, act='auto', init='xavier',
                 rnnType='gru', outputType='encoding'),
    srcLayers = 'rnn1',
    replace=True,
    name = 'rnn2'
)

conn.deepLearn.addLayer(
    model = 'gru',
    layer = dict(type='output', act='auto', init='xavier',
                 error='auto'),
    srcLayers = 'rnn2',
    replace=True,
    name = 'output'
)

conn.deepLearn.modelInfo(
    model='gru'
)
```

12. Train the Gated Unit Recurrent Neural Network with the same hyperparameters as the RNN.

```
conn.deepLearn.dlTrain(
    table    = dict(name = indata, where = '_PartInd_ = 1'),
    validTable = dict(name = indata, where = '_PartInd_ = 2'),
    target = 'dispute',
    inputs = 'complaint',
    texts = 'complaint',
    textParms = dict(initInputEmbeddings=dict(name='glove')),
    nominals = 'dispute',
    seed = '649',
    modelTable = 'gru',
    modelWeights = dict(name='gru_trained_weights', replace=True),
    optimizer = dict(miniBatchSize=100, maxEpochs=20,
            algorithm=dict(method='adam', beta1=0.9, beta2=0.999,
            learningRate=0.001, clipGradMax=100, clipGradMin=-100))
)
```

13. Use the dlScore action to score the GRU on the test data set.

```
conn.deepLearn.dlScore(
    table    = dict(name = indata, where = '_PartInd_ = 0'),
    model = 'gru',
    initWeights = 'gru_trained_weights',
    copyVars = 'dispute',
    textParms = dict(initInputEmbeddings=dict(name='glove')),
    casout = dict(name='gru_scored', replace=True)
)
```

    The GRU performed better (40%) than the RNN (46%). The addition of the gates to regulate the flow of information in this small data set help build a more accurate model.

14. Print the first five observations of the scored GRU model to view the variables available to compare model performance.

```
conn.table.fetch(table='gru_scored', to=5)['Fetch']
```

15. Download the results of the GRU to the client and create a histogram of the predicted probabilities.

```
gru_scored = conn.CASTable(name='gru_scored')
df = gru_scored.to_frame()

plt.figure(figsize=(8,8))
plt.hist(df['_DL_PredP_'], histtype='bar', ec='black')
plt.title('Predicted Probabilities', fontsize=20)
plt.xlabel('Probability', fontsize=15)
plt.ylabel('Frequency', fontsize=15)
plt.show()
```

    The histogram represents the predicted probabilities for the dispute on the test data. Most predictions were close to an estimated probability of 0.5 indicating the model had high uncertainty in scoring the data. Some predicted probabilities are near 1 indicating the model was highly confident predicting a dispute or note for some complaints in the test data.

16. Use the freq action from the simple action set to find the frequency of events for the test set. Print the proportion of non-events using open source syntax.

```
events = conn.simple.freq(
    table = dict(name = indata, where = '_PartInd_ = 0'),
    inputs = "dispute"
)['Frequency']

display(events)
print('Rate of Non-Events = ' + str(round(events['Frequency'][0]/
   events['Frequency'].sum(),4)))
```
The model did not simply predict the level with the largest frequency, indicating the model does in fact have some predictive power.

17. Use the crosstab action from the simple action set to find the cross-tabulation frequency matrix for the predicted vs actual events. Then compute the misclassification rate and the conditional probabilities of predicting a dispute given it is actually a dispute and predicting no dispute given it is actually not a dispute.
```
crosstab = conn.simple.crossTab(
    table = 'gru_scored',
    row = 'dispute', col = '_DL_PredName_'
)['Crosstab']

display(crosstab)
crosstab = crosstab.drop('dispute', axis=1)
print('Misclassification Rate = ' + str(round(1-
    pd.DataFrame(np.diagonal(crosstab)).sum(axis=0) /
    crosstab.values.sum(),4)[0]))

print('Predict Negative | Negative = ' +
   str(round(crosstab.divide(crosstab.sum(axis=1),
   axis=0)['Col1'][0],4)))
print('Predict Positive | Positive = ' +
   str(round(crosstab.divide(crosstab.sum(axis=1),
   axis=0)['Col2'][1],4)))
```
The conditional probabilities show it is easier to correctly predict no dispute than correctly predicting a dispute.

18. Use the assess action from the percentile action set and assess both the GRU and the RNN.
```
conn.loadActionSet('percentile')
actions = conn.builtins.help(actionSet='percentile')

conn.percentile.assess(
   table = "gru_scored",
   inputs = '_DL_P0_',
   casout = dict(name="gru_assess", replace=True),
   response = 'dispute',
   event = "1"
)

conn.percentile.assess(
   table = "rnn_scored",
   inputs = '_DL_P0_',
   casout = dict(name="rnn_assess", replace=True),
   response = 'dispute',
   event = "1"
)
```
19. Download the ROC tables to the client and combine them into one data frame.
```
gru_assess = conn.CASTable(name = "gru_assess_ROC")
gru_assess = gru_assess.to_frame()
gru_assess['Model'] = "GRU"

rnn_assess = conn.CASTable(name = "rnn_assess_ROC")
rnn_assess = rnn_assess.to_frame()
rnn_assess['Model'] = "RNN"

df_assess = pd.DataFrame()
df_assess = pd.concat([gru_assess, rnn_assess])
```
20. Print the confusion matrix for each model at a 0.50 cutoff rate.
```
cutoff_index = df_assess['_Cutoff_']==0.5
compare = df_assess[cutoff_index].reset_index(drop=True)
compare[['Model','_TP_','_FP_','_FN_','_TN_']]
```
21. Compare ROC curves for each model on the client.
```
plt.figure(figsize=(8,8))
plt.plot()
models = list(df_assess.Model.unique())

for X in models:
    tmp = df_assess[df_assess['Model']==X]
    plt.plot(tmp['_FPR_'],tmp['_Sensitivity_'], label=X+'
       (C=%0.2f)'%tmp['_C_'].mean())
```

```
plt.xlabel('False Positive Rate', fontsize=15)
plt.ylabel('True Positive Rate', fontsize=15)
plt.legend(loc='lower right')
plt.show()
```

Again, the GRU outperformed the RNN model and had a larger area under the curve when comparing ROC curves.

22. End the CAS session.

```
conn.session.endSession()
```