**Preprocessing Review**

Why do we **preprocess** data when we build machine learning pipelines?

We preprocess data for two principle reasons:

1. To transform the data to better suit a model's underlying assumptions.
2. To format the data in the way a model expects.

Today, we're concerned with this second reason.

**Inputs to Neural Networks**

What does the input to a neural network look like?

Inputs to neural networks are **vectors**. Each entry in the vector corresponds to a feature, which the net uses to make predictions.

Crucially, these vectors contain can contain only *numerical* data. They *cannot* contain string data.

## One-Hot Encoding

This poses a problem when we want to train a neural network on categorical data as we have in indeed\_job\_dataset.

The class columns contain values below:

**Queried\_Salary:**

array(['<80000', '80000-99999', '100000-119999', '120000-139999',

'140000-159999', '>160000'], dtype=object)

**Job\_Type:**

array(['data\_scientist', 'data\_analyst', 'data\_engineer'], dtype=object)

**Skill:**

array(["['SAP', 'SQL']",

"['Machine Learning', 'R', 'SAS', 'SQL', 'Python']",

"['Data Mining', 'Data Management', 'R', 'SAS', 'SQL', 'STATA', 'SPSS', 'Data Analysis', 'Python']",

...,

"['Spring', 'Data Management', 'Hadoop', 'Kafka', 'Java', 'Cassandra', 'S3', 'Spark', 'Kubernetes', 'CI', 'Design Experience', 'NoSQL', 'AWS']",

"['Spring', 'Ruby', 'Test Automation', 'Scripting', 'SDLC', 'Scala', 'Perl', 'Kafka', 'Management Experience', 'Ansible', 'Java', 'Docker', 'Python', 'Military Experience', 'Jenkins']",

"['JavaScript', 'TS/SCI Clearance', 'XML', 'Hadoop', 'HTML5', 'JSON', 'Java', 'Software Development', 'NoSQL', 'AJAX', 'CSS']"],

dtype=object)

**Location:**

array(['MO', 'TX', 'OR', 'DC', 'MD', 'NY', 'GA', 'ID', 'PA', 'FL', 'MA',

'VA', 'NJ', 'LA', 'CA', 'MN', 'WA', 'NC', 'IL', 'CO', 'UT', 'OH',

'USA', 'AR', 'ME', 'NV', 'CT', 'REMOTE', 'RI', 'TN', 'WI', 'SC',

'MI', 'KY', 'AZ', 'NE', 'IN', 'NM', 'AL', 'KS', 'IA', 'DE', nan,

'HI', 'NH', 'OK', 'VT', 'WV', 'SD', 'WY', 'MT', 'ND'], dtype=object)

As these are not numerical values, we can't use them to fit our neural network. To fix this, we must convert each class label to a numerical value.

We do this via the following steps:

1. **Label Encoding**. First, we convert the possible classes to integer labels. E.g., 'data\_analyst' will be 1; 'data\_engineer', 2; and 'data\_scientist', 3.
2. **One-Hot Encoding**. Then, we set each row's class value to an *array*. This array will have a 1 in whichever slot corresponds to the integer label. E.g., after one-hot encoding, a row with the class iris-setosa will have the array [1, 0, 0]. A row with class iris-virginica, the array [0, 0, 1]; etc.

In many cases, categories in the data sets you work with will already be label-encoded like we have for some skills and locations label encoded columns. In this case, we applied one-hot encoding to all string columns.

LET’S START!

It is really important to scale our data before using multilayer perceptron models.

Without scaling, it is often difficult for the training cycle to converge.

labels=["Job\_Type","Job\_Title","Company"]

X = df.drop(labels=labels, axis=1)

y = df["Job\_Type"]

print(X.shape, y.shape)

(5715, 925) (5715,)

1. **Scaling**

We use MinMaxScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder, MinMaxScaler

from tensorflow.keras.utils import to\_categorical

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, random\_state=1)

print(X\_train.shape,X\_test.shape,y\_train.shape,y\_test.shape)

(4286, 925) (1429, 925) (4286,) (1429,)

X\_scaler = MinMaxScaler().fit(X\_train)

X\_train\_scaled = X\_scaler.transform(X\_train)

X\_test\_scaled = X\_scaler.transform(X\_test)

1. **One-hot encode the labels**

label\_encoder = LabelEncoder()

label\_encoder.fit(y\_train)

encoded\_y\_train = label\_encoder.transform(y\_train)

encoded\_y\_test = label\_encoder.transform(y\_test)

1. **Creating our Model**

We must first decide what kind of model to apply to our data.

For numerical data, we use a regressor model.

For categorical data, we use a classifier model.

In our project, we use a classifier to build our network and

defining our Model Architecture (the layers), we first need to create a sequential model.

# Create model and add layers

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

model = Sequential()

Next, we add our first layer. This layer requires you to specify both the number of inputs and the number of nodes that you want in the hidden layer.

model.add(Dense(units=10, activation='relu', input\_dim=X\_features))

units 🡺 number of hidden nodes

x\_features 🡺 number of our inputs

we add second layer

model.add(Dense(units=10, activation='relu'))

Our final layer is the output layer. Here, we need to specify the activation function ( softmax is our classification) and the number of classes (labels) that we are trying to predict (y\_features).

model.add(Dense(units=y\_features, activation='softmax'))

y\_features 🡺 Job\_Types

1. **Compile the model**

Now that we have our model architecture defined, we must compile the model using a loss function and optimizer. We can also specify additional training metrics such as accuracy.

We use categorical crossentropy for categorical data and mean squared error for regression

Hint: your output layer in this example is using software for logistic regression (categorical)

model.compile(optimizer='adam',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

1. **Model Summary**

model.summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 10) 9260

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 110

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 3) 33

=================================================================

Total params: 9,403

Trainable params: 9,403

Non-trainable params: 0

1. **Training the model**

Finally, we train our model using our training data

Training consists of updating our weights using our optimizer and loss function. In our project, we choose 20 iterations (loops) of training that are called epochs.

We also choose to shuffle our training data and increase the detail printed out during each training cycle.

model.fit(

    X\_train\_scaled,

    y\_train\_categorical,

    epochs=20,

    shuffle=True

)

Epoch 1/20

134/134 [==============================] - 0s 1ms/step - loss: 0.9538 - accuracy: 0.5343

Epoch 2/20

134/134 [==============================] - 0s 1ms/step - loss: 0.6043 - accuracy: 0.7928

Epoch 3/20

134/134 [==============================] - 0s 1ms/step - loss: 0.4468 - accuracy: 0.8327

Epoch 4/20

134/134 [==============================] - 0s 1ms/step - loss: 0.3782 - accuracy: 0.8560

Epoch 5/20

134/134 [==============================] - 0s 1ms/step - loss: 0.3375 - accuracy: 0.8745

Epoch 6/20

134/134 [==============================] - 0s 1ms/step - loss: 0.3079 - accuracy: 0.8843

Epoch 7/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2852 - accuracy: 0.8962

Epoch 8/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2673 - accuracy: 0.8976

Epoch 9/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2511 - accuracy: 0.9046

Epoch 10/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2394 - accuracy: 0.9085

Epoch 11/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2299 - accuracy: 0.9113

Epoch 12/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2202 - accuracy: 0.9162

Epoch 13/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2112 - accuracy: 0.9197

Epoch 14/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2049 - accuracy: 0.9246

Epoch 15/20

134/134 [==============================] - 0s 1ms/step - loss: 0.2014 - accuracy: 0.9230

Epoch 16/20

134/134 [==============================] - 0s 1ms/step - loss: 0.1930 - accuracy: 0.9279

Epoch 17/20

134/134 [==============================] - 0s 1ms/step - loss: 0.1881 - accuracy: 0.9293

Epoch 18/20

134/134 [==============================] - 0s 1ms/step - loss: 0.1850 - accuracy: 0.9342

Epoch 19/20

134/134 [==============================] - 0s 1ms/step - loss: 0.1805 - accuracy: 0.9323

Epoch 20/20

134/134 [==============================] - 0s 1ms/step - loss: 0.1752 - accuracy: 0.9370

<tensorflow.python.keras.callbacks.History at 0x7f1746587e80>

1. **Quantify the model**

We use our testing data to validate our model. This is how we determine the validity of our model (i.e. the ability to predict new and previously unseen data points)

model\_loss, model\_accuracy = model.evaluate(

    X\_test\_scaled, y\_test\_categorical, verbose=2)

print(

    f"Normal Neural Network - Loss: {model\_loss}, Accuracy: {model\_accuracy}")

45/45 - 0s - loss: 0.3214 - accuracy: 0.8929

Normal Neural Network - Loss: 0.32138586044311523, Accuracy: 0.892932116985321

1. **Making Predictions with the new data**

We use our trained model to make predictions using `model.predict`

predictions = model.predict\_classes(X\_test\_scaled)

array([2, 0, 2, ..., 1, 1, 2])

y\_test\_categorical

array([[0., 0., 1.],

[1., 0., 0.],

[0., 0., 1.],

...,

[0., 1., 0.],

[0., 1., 0.],

[0., 0., 1.]], dtype=float32)

true\_labels = label\_encoder.inverse\_transform(predictions)

array(['data\_scientist', 'data\_analyst', 'data\_scientist', ...,

'data\_engineer', 'data\_engineer', 'data\_scientist'], dtype=object)

print(f"Predicted classes: {true\_labels[:5]}")

print(f"Actual Labels: {list(y\_test[:5])}")

Predicted classes: ['data\_scientist' 'data\_analyst' 'data\_scientist' 'data\_scientist'

'data\_engineer']

Actual Labels: ['data\_scientist', 'data\_analyst', 'data\_scientist', 'data\_analyst', 'data\_engineer']