HW 4: Text Models & Neural Networks

Submitted by: Gideon Tay
My UNI: gt2528
Contact me at: gideon.tay@columbia.edu

Part A: Build a classification model using text data

Let's first import all the required libraries for Part A:

In [ ]:
```python
# Import libraries required for Part A
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import f1_score, classification_report
```

A1. Import the data. The headlines will become your vectorized X matrix, and the labels indicate a binary classification (clickbait or not).

In [2]:
```python
# Import data from local directory
df = pd.read_csv('text_training_data.csv')

# Display the first 5 rows of the data
df.head(5)
```

Out[2]:

	headline	label
0	MyBook Disk Drive Handles Lots of Easy Backups	not clickbait
1	CIT Posts Eighth Loss in a Row	not clickbait
2	Candy Carson Singing The "National Anthem" Is ...	clickbait
3	Why You Need To Stop What You're Doing And Dat...	clickbait
4	27 Times Adele Proved She's Actually The Reale...	clickbait

A2. Convert the headline data into an X feature matrix using a simple bag of words approach.

We convert the `label` column to a binary variable where 1 is clickbait, and 0 is not clickbait. Then, we use `CountVectorizer()` to apply a simple bag of words approach to transform the data in `headline` into an X feature matrix:

In [3]:
```python
# Extract the 'headline' and 'label' columns
headlines = df['headline']
labels = df['label']

# Convert labels to binary (1 for clickbait, 0 for not clickbait)
y = labels.map({'clickbait': 1, 'not clickbait': 0})

# Vectorize the headline data using a Bag-of-Words approach
vect = CountVectorizer()
X = vect.fit_transform(headlines)

# Explore the resulting matrix
print(f"Feature matrix shape: {X.shape}")
print(f"Number of headlines: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"The sparse matrix encoding:\n{X.toarray()}")
feature_names = vect.get_feature_names_out()
print(f"Every 1000th feature:\n{feature_names[::1000]}")
```

```
Feature matrix shape: (24979, 20332)
Number of headlines: 24979
Number of features: 20332
The sparse matrix encoding:
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]]
Every 1000th feature:
['00' 'anaconda' 'begins' 'camera' 'compostela' 'deliver' 'electroputere'
 'flats' 'gromit' 'ignores' 'kershner' 'mainframes' 'movement' 'overwatch'
 'predators' 'relieved' 'screening' 'spadafora' 'tartan' 'undecided'
 'wizard']
```

The X feature matrix shape suggests that there are 24979 headlines and 20332 features, with each feature corresponding to a unique word.

The array shows the number of times each unique word appears in a given headline. Naturally, most of the values would be 0 (thus the name 'sparse matrix encoding'), since each headline would only have a small amount of words compared to the 20332 unique words in our feature set.

I have also printed out every 1000th feature out of the 20332 features, to give you a small sample of features (unique words) in our feature set.

A3. Run logistic regression to predict clickbait headlines. Remember to train_test_split your data and use GridSearchCV to find the best value of C. You should evaluate your data with F1 scoring.

Let's first split the data into training and testing sets:

```
In [4]:
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
    )

# Confirm the split sizes
print(f"Training data shape (features): {X_train.shape}")
print(f"Training data shape (target): {y_train.shape}")
print(f"Testing data shape (features): {X_test.shape}")
print(f"Testing data shape (target): {y_test.shape}")
```

```
Training data shape (features): (19983, 20332)
Training data shape (target): (19983,)
Testing data shape (features): (4996, 20332)
Testing data shape (target): (4996,)
```

Next, set up a logistic regresssion model and use `GridSearchCV()` to find the best value for parameter `C`:

```
In [5]:
# Set up the logistic regression model and GridSearchCV
log_reg = LogisticRegression(solver='liblinear')  # Use liblinear for small dataset
params = {'C': [0.01, 0.1, 1, 10, 100]}  # Values for regularization strength

# Use F1 score for GridSearchCV, statified 5-fold cross validation
grid = GridSearchCV(log_reg, params, cv=5, scoring='f1')
grid.fit(X_train, y_train)

# Print best parameter and cross-validation score
print(f"Best cross-validation score: {round(grid.best_score_,2)}")
print(f"Best value for C: {grid.best_params_['C']}")
```

```
Best cross-validation score: 0.97
Best value for C: 100
```

Lastly, evaluate the logistic regression model that uses the best parameter value, using an f1 score and the test set:

```
In [6]:
# Evaluate the best model on the test set
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
f1 = f1_score(y_test, y_pred)
print(f"F1 Score on the test set: {round(f1,2)}")

# Print a detailed classification report
print("\nClassification Report:")
print(classification_report(
    y_test, y_pred, target_names=['Not Clickbait', 'Clickbait']
    ))
```

```
F1 Score on the test set: 0.97

Classification Report:
               precision    recall  f1-score   support

Not Clickbait       0.97      0.97      0.97      2610
    Clickbait       0.97      0.96      0.97      2386

     accuracy                           0.97      4996
    macro avg       0.97      0.97      0.97      4996
 weighted avg       0.97      0.97      0.97      4996
```

A4. Run 2 more logistic regression models by changing the vectorization approach (e.g. using n-grams, stop_words, and other techniques we discussed). In both cases, keep your logistic regression step the same. Only change how you're generating the X matrix from the text data.

Model 2: Unigram + Bigram

For our first vectorization approach, let us consider both unigrams and bigrams. This means that we consider sets of 2 words as features as well, beyond single words. This could be useful since sometimes words' meanings are only derived not alone but in pairs. As you can see below, our feature set increases greatly, from 20,332 to 135,950, and there are both single and double word features:

In [7]:
```python
# New Approach: Unigram + Bigram
vect2 = CountVectorizer(ngram_range=(1, 2))
X2 = vect2.fit_transform(headlines)

# Explore the resulting matrix
print(f"Feature matrix shape: {X2.shape}")
print(f"Number of features: {X2.shape[1]}")
feature_names2 = vect2.get_feature_names_out()
print(f"Every 5000th feature:\n{feature_names2[::5000]}")
```

```
Feature matrix shape: (24979, 135950)
Number of features: 135950
Every 5000th feature:
['00' 'about talking' 'and israel' 'award belonging' 'boris'
 'celtics keep' 'copies' 'diet' 'especially for' 'fluffy puppies'
 'gets 10' 'held in' 'in monster' 'jenner received' 'life hacks'
 'men shirt' 'new indictments' 'on debt' 'permitted' 'purchase mobile'
 'rights victory' 'sharp decline' 'steak and' 'that made' 'to defeat'
 'uninjured after' 'when eddie' 'young indians']
```

Now, let's evaluate the model. Since we will have to repeat this process again for the next model, we define a function so we can re-use it later,

In [8]:
```python
def split_gridSearch_evaluate(X, y):
    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
        )

    # Use GridSearchCV to find the best parameter
    grid = GridSearchCV(log_reg, params, cv=5, scoring='f1')
    grid.fit(X_train, y_train)

    # Get best parameter and cross-validation score
    best_cross_val_score = round(grid.best_score_,3)
    best_C = grid.best_params_['C']

    # Evaluate the best model on the test set
    best_model = grid.best_estimator_
    y_pred = best_model.predict(X_test)
    f1 = round(f1_score(y_test, y_pred),3)

    # Return key values
    return {
        "best_cross_val_score": best_cross_val_score,
        "best_C": best_C,
        "f1_score": f1,
        "best_model": best_model
    }

# Print results
results2 = split_gridSearch_evaluate(X2, y)
print(f"Best cross-validation score: {results2["best_cross_val_score"]}")
print(f"Best value for C: {results2["best_C"]}")
print(f"F1 Score on the test set: {results2["f1_score"]}")
```

```
Best cross-validation score: 0.972
Best value for C: 100
F1 Score on the test set: 0.968
```

Model 3: remove infrequent + stop words

For our second new vectorization approach, remove `stop_words` and words that appear in less than 2 headlines. Stop words are words that do not have much meaning and are thus not useful to aid in prediction. Removing `stop_words` and infrequent words reduces our feature set from 20,332 to 10,984.

In [9]:
```
# New Approach: Remove infrequent and stop words
vect3 = CountVectorizer(min_df=2, stop_words="english")
X3 = vect3.fit_transform(headlines)

# Explore the resulting matrix
print(f"Feature matrix shape: {X3.shape}")
print(f"Number of features: {X3.shape[1]}")
feature_names3 = vect3.get_feature_names_out()
print(f"Every 1000th feature:\n{feature_names3[::1000]}")
```

```
Feature matrix shape: (24979, 10984)
Number of features: 10984
Every 1000th feature:
['00' 'base' 'climbing' 'doc' 'fred' 'inexpensive' 'mainstream'
 'overturns' 'recruit' 'sirius' 'toddlers']
```

Now, let's evaluate the model, using the `split_gridSearch_evaluate()` function we defined earlier:

In [10]:
```
# Print results
results3 = split_gridSearch_evaluate(X3, y)
print(f"Best cross-validation score: {results3["best_cross_val_score"]}")
print(f"Best value for C: {results3["best_C"]}")
print(f"F1 Score on the test set: {results3["f1_score"]}")
```

```
Best cross-validation score: 0.948
Best value for C: 10
F1 Score on the test set: 0.945
```

A5. Which of your 3 models performed best? What are the most significant coefficients in each, and how do they compare?

Let's summarize the F1 scores each model has against the test set in a dataframe:

In [11]:
```
# Create a dictionary for the DataFrame
f1_score_results = {
    'Model': ['Model 1', 'Model 2', 'Model 3'],
    'Description': [
        'simple bag-of-words',
        'unigram + bigram',
        'remove infrequent + stop words'
        ],
    'Test set F1 score': [round(f1,3), results2["f1_score"], results3["f1_score"]]
}

# Create and display the DataFrame
f1_score_results_table = pd.DataFrame(f1_score_results)
f1_score_results_table
```

Out[11]:

	Model	Description	Test set F1 score
0	Model 1	simple bag-of-words	0.966
1	Model 2	unigram + bigram	0.968
2	Model 3	remove infrequent + stop words	0.945

Model 1 and 2 have similar performance, though model 2 which also considers bigrams beyond unigrams performs marginally better. Model 3 is the worst-performing model with the lowest F1 score.

Now let's observe the most significant coefficients in each model. For each model, we only consider the one with the best parameter `C` value based on `GridSearchCV` conducted earlier. Then, we extract the top 10 coefficients with the highest absolute value, along with their corresponding feature names, and display them in a table:

In [12]:
```
# Function to extract top 10 coefficients from a model and vectorizer
def get_top_coefficients(best_model, vect):
    coefficients = best_model.coef_.flatten()
    feature_names = vect.get_feature_names_out()

    # Pair coefficients with feature names and sort
    coef_with_terms = sorted(
        zip(feature_names, coefficients), # Create list of tuples of (coeff, featur
        key=lambda x: abs(x[1]), # Sort by absolute values of coefficients
        reverse=True # So arranged from highest to lowest
        )

    # Return top 10 coefficients and feature names only
    top_10 = coef_with_terms[:10]
    df = pd.DataFrame(top_10, columns=['Feature', 'Coefficient'])
    return df

# Extract top 10 coefficients for each model and store in DataFrames
top_coeffs_model1 = get_top_coefficients(best_model, vect)
top_coeffs_model2 = get_top_coefficients(results2['best_model'], vect2)
top_coeffs_model3 = get_top_coefficients(results3['best_model'], vect3)

# Rename the 'Feature' and 'Coefficient' columns for each model
top_coeffs_model1 = top_coeffs_model1.rename(
    columns={'Feature': 'Model 1 Feature', 'Coefficient': 'Model 1 Coefficient'})
top_coeffs_model2 = top_coeffs_model2.rename(
    columns={'Feature': 'Model 2 Feature', 'Coefficient': 'Model 2 Coefficient'})
top_coeffs_model3 = top_coeffs_model3.rename(
    columns={'Feature': 'Model 3 Feature', 'Coefficient': 'Model 3 Coefficient'})

# Merge the DataFrames side by side and display them
df_comparison = pd.concat([
    top_coeffs_model1,
    top_coeffs_model2,
    top_coeffs_model3
], axis=1)
df_comparison
```

Out[12]:

	Model 1 Feature	Model 1 Coefficient	Model 2 Feature	Model 2 Coefficient	Model 3 Feature	Model 3 Coefficient
0	ferry	-9.294036	you	6.392502	guess	5.343873
1	pitch	-8.514286	your	6.300404	kills	-4.484662
2	you	7.289243	confessions	5.858850	2015	4.430760
3	2015	7.047258	this	5.836613	crossword	4.372021
4	buzzfeed	6.966647	these	5.467637	know	4.319080
5	victories	6.586090	but not	5.417677	zealand	-4.258099
6	confessions	6.539295	21	5.109011	guy	4.244425
7	these	6.431714	2015	4.953683	sucked	4.243858
8	your	6.368054	17	4.935882	inauguration	-4.140832
9	feminist	6.191835	buzzfeed	4.804098	character	4.099228

Some observations:

- Across all 3 models, the most significant coefficients have absolute values between 4 and 10.
- Model 2's features with the top coefficients are 'you' and 'your', and these two words appear in Model 1's top 10 list as well. However, 'you' and 'your' may be stop words removed in model 3. Perhaps, such words have good predictive value of whether a headline is clickbait, as clickbait headlines may tend to address the reader directly. Then, the traditional stop word list may not be most appropriate. Indeed, we previously observed that model 3 has the lowest F1 score among the 3 models.
- Model 2 has a bigram 'but not' in the top 10. Such a feature is not possible in the other 2 models, which don't allow for bigrams.
- Interestingly, we only find some, but limited overlaps in the top predictive features across the 3 models, despite them using the same underlying data. This shows that the choice of the vectorization approach when producing feature sets from text data heavily affects the model.

Part B: Build a Predictive Neural Network Using Keras

Let's first import all the libraries needed in Part B:

In [13]:
```
# Import libraries required for Part B (not yet imported in Part A)
import tensorflow.keras as keras
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.preprocessing import LabelEncoder, StandardScaler
```

B1. Import and load the iris dataset

In [14]:
```
# Import dataset from url
url = "http://vincentarelbundock.github.io/Rdatasets/csv/datasets/iris.csv"
data = pd.read_csv(url)

# Remove the 'rownames' column (no useful information)
data = data.loc[:, data.columns != 'rownames']

# Display the first 5 rows
data.head(5)
```

Out[14]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

B2. Using the Sequential interface in Keras, build a model with 2 hidden layers with 16 neurons in each. Compile and fit the model. Assess its performance using accuracy on data that has been train_test_split.

Let us first pre-process the data by splitting it into features (X) and target (y), one-hot encoding the target, and scaling the features. Then, split the data into training and test sets:

In [15]:
```
# Prepare features and target
X = data.loc[:, data.columns != 'Species'] # Drop Species column
y = data['Species'] # Target column is 'Species'

# Encode species names into numeric labels to pass into to_categorical
label_encoder = LabelEncoder()
y_int = label_encoder.fit_transform(y)  # Converts to [0, 1, 2]

# One-hot encode the y data using to_categorical(), 3 categories
y_encoded = keras.utils.to_categorical(y_int, num_classes=3)

# Standardize the features, may improve predictive accuracy
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_encoded, test_size=0.2, random_state=42
    )

# Print some information about our dataset
print(f"Training samples: {X_train.shape[0]}")
print(f"Test samples: {X_test.shape[0]}")
print(f"No. of features: {X_train.shape[1]}")
```

```
Training samples: 120
Test samples: 30
No. of features: 4
```

The key variable of interest which we want to predict is `Species`. To better understand this classification problem, let us first observe all the possible unique values of `Species` in our dataset:

In [16]:
```
# List unique values in 'Species'
unique_values = data['Species'].unique()
print(unique_values)
```

```
['setosa' 'versicolor' 'virginica']
```

Since there are only 3 possible species, our model's output layer will have 3 categories. Also note that we have 4 predictor variables: `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`.

Below, we build a model with 4 inputs (our predictor variables), 2 hidden layers with 16 nodes each, and 1 output layer with 3 categories. We use the `relu` activation function for the hidden layers, and the `softmax` function for the output layer, which is used to calculate 0 to 1 probabilities for each of the 3 categories.

In [17]:
```
# Build model using Sequential interface in Keras
model = Sequential([
    Dense(16, input_shape=(4,)), # 4 inputs; 16 nodes in hidden layer 1
    Activation('relu'), # relu for hidden layer 1
    Dense(16), # 16 nodes in hidden layer 2
    Activation('relu'), # relu for hidden layer 2
    Dense(3), # 3 categories in output layer
    Activation('softmax')
])

# View the model summary
model.summary()
```

```
c:\Users\gideo\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWa
rning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequ
ential models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                   │ (None, 16)             │            80 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ activation (Activation)         │ (None, 16)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 16)             │           272 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ activation_1 (Activation)       │ (None, 16)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 3)              │            51 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ activation_2 (Activation)       │ (None, 3)              │             0 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 403 (1.57 KB)
 Trainable params: 403 (1.57 KB)
 Non-trainable params: 0 (0.00 B)
```

Having built and specified the model's structure, we can now compile and fit the model. Compiling a model enables us to configure its learning process. Here, we use a stochastic gradient descent optimizer and specify loss as `categorical_crossentropy` since we have 3 possible categories of `Species`. After compiling, we fit our model with the training data:

In [18]:
```python
# Compile the model: configure its learning process
model.compile(loss='categorical_crossentropy', # Since there are 3 categories
              optimizer='sgd', # Stochastic gradient descent
              metrics=['accuracy'])

# Fit the model
model.fit(X_train, y_train, epochs=20, batch_size=128)
```

```
Epoch 1/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 513ms/step - accuracy: 0.2417 - loss: 1.3630
Epoch 2/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.2500 - loss: 1.3327
Epoch 3/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.2417 - loss: 1.3040
Epoch 4/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.2667 - loss: 1.2769
Epoch 5/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.2667 - loss: 1.2511
Epoch 6/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.2750 - loss: 1.2267
Epoch 7/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.2750 - loss: 1.2034
Epoch 8/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.2750 - loss: 1.1813
Epoch 9/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3083 - loss: 1.1602
Epoch 10/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3500 - loss: 1.1401
Epoch 11/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.3667 - loss: 1.1209
Epoch 12/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.3667 - loss: 1.1025
Epoch 13/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3833 - loss: 1.0849
Epoch 14/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.4000 - loss: 1.0681
Epoch 15/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.4250 - loss: 1.0520
Epoch 16/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.4583 - loss: 1.0365
Epoch 17/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.4750 - loss: 1.0217
Epoch 18/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.5000 - loss: 1.0075
Epoch 19/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.5000 - loss: 0.9939
Epoch 20/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.5333 - loss: 0.9808
```

```
Out[18]: <keras.src.callbacks.history.History at 0x296b4d102c0>
```

Finally, we can assess the model's performance on the test data:

```
In [19]: # Evaluate the model's performance
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 1.000
Test Accuracy: 0.567
```

B3. Run 2 additional models using different numbers of hidden layers and/or hidden neurons

Let's build, compile, fit, and evaluate the performance of a model with 3 hidden layers instead of 2, while keeping the number of neurons per hidden layer (16) the same:

In [20]:
```
# Build model using Sequential interface in Keras
model2 = Sequential([
    Dense(16, input_shape=(4,)), # 4 inputs; 16 nodes in hidden layer 1
    Activation('relu'), # relu for hidden layer 1
    Dense(16), # 16 nodes in hidden layer 2
    Activation('relu'), # relu for hidden layer 2
    Dense(16), # 16 nodes in hidden layer 3
    Activation('relu'), # relu for hidden layer 3
    Dense(3), # 3 categories in output layer
    Activation('softmax')
])

# Compile the model: configure its learning process
model2.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# Fit the model
model2.fit(X_train, y_train, epochs=20, batch_size=128, verbose=0)

# Evaluate the model's performance
score2 = model2.evaluate(X_test, y_test, verbose=0)
print("Test loss for model 2: {:.3f}".format(score2[0]))
print("Test Accuracy for model 2: {:.3f}".format(score2[1]))
```

```
Test loss for model 2: 0.886
Test Accuracy for model 2: 0.867
```

Let's build, compile, fit, and evaluate the performance of a model with 32 neurons per hidden layer instead of 16, while keeping the number of hidden layers (2) the same as our initial model:

In [21]:
```
# Build model using Sequential interface in Keras
model3 = Sequential([
    Dense(32, input_shape=(4,)),
    Activation('relu'),
    Dense(32),
    Activation('relu'),
    Dense(3),
    Activation('softmax')
])

# Compile the model: configure its learning process
model3.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# Fit the model
model3.fit(X_train, y_train, epochs=20, batch_size=128, verbose=0)

# Evaluate the model's performance
```

```
score3 = model3.evaluate(X_test, y_test, verbose=0)
print("Test loss for model 3: {:.3f}".format(score3[0]))
print("Test Accuracy for model 3: {:.3f}".format(score3[1]))
```

```
Test loss for model 3: 1.112
Test Accuracy for model 3: 0.233
```

B4. How does the performance compare between your 3 models?

Let's summarize the test results (both loss and accuracy) for each of the 3 models in a dataframe:

In [22]:
```
# Create a dictionary for the DataFrame
test_result = {
    'Model': ['Model 1', 'Model 2', 'Model 3'],
    'Hidden layers': [2, 3, 2],
    'Hidden neurons per layer': [16, 16, 32],
    'Loss': [score[0], score2[0], score3[0]],
    'Accuracy': [score[1], score2[1], score3[1]]
}

# Create and display the DataFrame
df = pd.DataFrame(test_result)
df
```

Out[22]:

	Model	Hidden layers	Hidden neurons per layer	Loss	Accuracy
0	Model 1	2	16	1.000244	0.566667
1	Model 2	3	16	0.886357	0.866667
2	Model 3	2	32	1.112339	0.233333

Accuracy measures the proportion of correct predictions out of the total number of predictions, while loss quantifies the error for each prediction and aggregates it into a single number. Here, we use the categorical cross-entropy loss function.

Model 2 has the highest accuracy and lowest loss among the 3 models.This suggests increasing the number of hidden layers improves model performance in our example.

Model 3 has the lowest accuracy and highest loss among the 3 models. This suggests increasing the number of hidden neurons per layer does not improve model performance in our example.