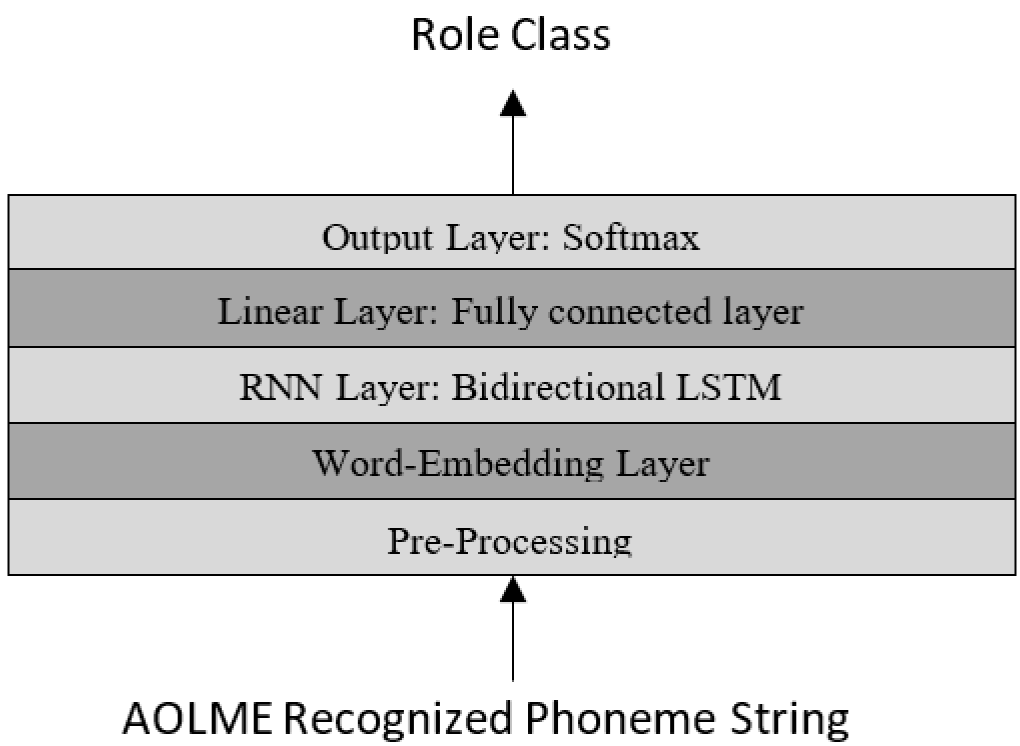
Roles Classifier

The model implemented here follows the following architecture diagram:



We start with the import declarations:

In [16]:
```
from collections import Counter
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
from BiLstmClassifier import BiLstmFixedLength, BiLstmVariableLength, Bi

import numpy as np
import pandas as pd
import spacy
import torch
import torch.nn.functional as F
```

0. Hyper Parameters Definition

Defining the correct hyper-parameters to refine the performance of the model is very important. We must consider some situations around the values to be assigned.

0.1 Epochs

We must define a correct number of epochs to allow the model to learn from the dataset. If this is to few, the model will not learn anything, if the value is too high, we can fall into over-fitting or wasting processing time where the model can't learn anymore.

0.2 Batch Size

It defines the amount of data for each batch to be used on each training epoch. Because we have no much data, the value here is small.

0.3 Embedded Layer Dimension

Word embeddings are always around 50 and 300 in length, longer embedding vectors don't add enough information and smaller ones don't represent the semantics well enough. For this model, we are using small sentences for most of the cases.

0.4 Hidden Layer Dimension

This parameter represents how complex is the language used in the conversations. For example, if the sentences belong to a literature book from Shakespeare, it probably will use a sophisticated language, and we can assign a value of 70. On the other hand, if the sentences belong to simple chat talking about movies, it is maybe simpler, and we can assign a value of 30.

In [17]:
```
EPOCHS = 100
BATCH_SIZE = 1  # Small batches because the dataset is not bigger than
HIDDEN_LAYER_DIM = 60  # AOLME is not too complex language, it represen
EMBEDDED_LAYER_DIM = 50
```

1. Load dataset

We are going to use the dataset generated by the Jupyter Notebook "AOLME Datasets Generator" (main.ipynb)

In [18]:
```
roles = pd.read_csv('output/balanced_372.csv')
print(f'Dataset Size: {roles.shape}\n')
print(roles.head())
```

```
Dataset Size: (372, 2)

      Role                           Text
0  Student          you like how its like
1  Student  its like youre obsessed with
2  Student                   and this one
3  Student                             no
4  Student           i dont like summary
```

2. Pre-Processing

2.1. Mapping 'Roles' labels to numbers for vectorization

In [19]:
```
mapping = {'Student': 0, 'Co-Facilitator': 1, 'Facilitator': 2}
roles['Role'] = roles['Role'].apply(lambda x: mapping[x])
roles.head()


# Load English words model package
tok = spacy.load('en')


def tokenize(text: str):
    """
    This method tokenizes a sentence, considering the text is already lo
    ASCII, and  punctuation has been removed
    :param text: The sentence to be tokenized
    :return: A list containing each word of the sentence
    """
    return [token.text for token in tok.tokenizer(text)]
```

2.2. Dataset cleaning and Sentence Vectorizing

In [20]:
```python
# Count number of occurrences of each word
counts = Counter()
for index, row in roles.iterrows():
    counts.update(tokenize(row['Text']))

# Deletes words appearing only once
print(f'Number of Words before cleaning: {len(counts.keys())}')
for word in list(counts):
    if counts[word] < 2:
        del counts[word]
print(f'Number of Words after cleaning: {len(counts.keys())}\n')

# Creates vocabulary
vocab2index = {'': 0, 'UNK': 1}
words = ['', 'UNK']
for word in counts:
    vocab2index[word] = len(words)
    words.append(word)


def encode_sentence(text, vocabulary_map, n=70):
    """
    Encodes the sentence into a numerical vector, based on the vocabular
    :param text: The sentence
    :param vocabulary_map: A map assigning a number to each word in the
    :param n: Required vector size
    :return: Vectorized sentence and length
    """
    tokenized = tokenize(text)
    vectorized = np.zeros(n, dtype=int)
    enc1 = np.array([vocabulary_map.get(w, vocabulary_map["UNK"]) for w
    length = min(n, len(enc1))
    vectorized[:length] = enc1[:length]
    return vectorized, length


# Creates a new column into Dataset: each sentence expressed as a numer
roles['Vectorized'] = roles['Text'].apply(lambda x: np.array(encode_sent
print(roles.head())
```

```
Number of Words before cleaning: 662
Number of Words after cleaning: 347

   Role                              Text  \
0     0          you like how its like
1     0  its like youre obsessed with
2     0                    and this one
3     0                              no
4     0           i dont like summary

                                          Vectorized
0  [[2, 3, 4, 5, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
1  [[5, 3, 2, 1, 1, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
2  [[7, 8, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
3  [[10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
4  [[11, 12, 13, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
```

```
<ipython-input-20-5b2887f873d4>:38: VisibleDeprecationWarning: Creatin
g an ndarray from ragged nested sequences (which is a list-or-tuple of
lists-or-tuples-or ndarrays with different lengths or shapes) is depre
cated. If you meant to do this, you must specify 'dtype=object' when c
reating the ndarray
  roles['Vectorized'] = roles['Text'].apply(lambda x: np.array(encode_
sentence(x, vocab2index)))
```

Check if the dataset is balanced

In [21]:
```
Counter(roles['Role'])
```

Out[21]:
```
Counter({0: 124, 1: 124, 2: 124})
```

2.3 Split into training and validation partitions

In [22]:
```
X = list(roles['Vectorized'])
y = list(roles['Role'])
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.


class RolesDataset(Dataset):
    """
    Simple PyTorch Dataset wrapper defined by an array of vectorized ser
    """
    def __init__(self, input_x, input_y):
        self.X = input_x
        self.y = input_y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return torch.from_numpy(self.X[idx][0].astype(np.int32)), self.y


training_ds = RolesDataset(X_train, y_train)
validation_ds = RolesDataset(X_valid, y_valid)
```

2.4 Training and Validation Functions

In [23]:
```
def train_model(input_model, epochs=10, lr=0.001, verbose=True):
    """
    Trains the input model
    :param verbose: Prints each batch iteration
    :param input_model: Input Model
    :param epochs: The number of training epochs
    :param lr: Learning Rate
    :return: training loss, validation loss, validation accuracy, and va
    """
    parameters = filter(lambda p: p.requires_grad, input_model.parameter
    optimizer = torch.optim.Adam(parameters, lr=lr)

    for i in range(epochs):
        input_model.train()
        sum_loss = 0.0
        total = 0

        # Iterates on Training DataLoader
        for x, y, l in training_dl:
            x = x.long()
            y = y.long()
            y_pred = input_model(x, l)
            optimizer.zero_grad()
            loss = F.cross_entropy(y_pred, y)
            loss.backward()
            optimizer.step()
            sum_loss += loss.item() * y.shape[0]
            total += y.shape[0]

        val_loss, val_acc, val_rmse = get_metrics(input_model, validatio

        if verbose and (i + 1) % 20 == 1:
            print(f"Epoch {i}: training loss %.3f, valid. loss %.3f, val
                sum_loss / total, val_loss, val_acc, val_rmse))

    print(f"FINAL: training loss %.3f, valid. loss %.3f, valid. accuracy
        sum_loss / total, val_loss, val_acc, val_rmse))

    return sum_loss / total, val_loss, val_acc, val_rmse


def get_metrics(input_model, valid_dl):
    """
    Obtains current validation metrics
    :param input_model: Input Model
    :param valid_dl: Validation PyTorch DataLoader
    :return:
    """
    input_model.eval()
    correct = 0
    total = 0
    sum_loss = 0.0
    sum_rmse = 0.0

    # PyTorch uses CrossEntropy function to implement Softmax on the sa
    for x, y, l in valid_dl:
```

```
        x = x.long()
        y = y.long()
        y_hat = input_model(x, l)
        loss = F.cross_entropy(y_hat, y)
        pred = torch.max(y_hat, 1)[1]
        correct += (pred == y).float().sum()
        total += y.shape[0]
        sum_loss += loss.item() * y.shape[0]
        sum_rmse += np.sqrt(mean_squared_error(pred, y.unsqueeze(-1))) *
    return sum_loss / total, correct / total, sum_rmse / total
```

In [24]:

```
vocab_size = len(words)
training_dl = DataLoader(training_ds, batch_size=BATCH_SIZE, shuffle=Tru
validation_dl = DataLoader(validation_ds, batch_size=BATCH_SIZE)
```

BiLSTM - Fixed Length Input

We can see the implemented model class in BiLstmClassifier class (BiLstmClassifier.py).

BiLstmFixedLength has the following features:

- Word-Embedding Layer. # Embeddings: Vocabulary Size, Embeddings size: 50
- Bi-directional LSTM Layer. Input size: 50, Hidden size: 60
- Linear Layer. Fully connected layer, Input size: 60, 3 output features (roles)
- Dropout: 0.7
- Fixed Length Input (see encode_sentence function)

In [25]:
```
model_fixed = BiLstmFixedLength(vocab_size, EMBEDDED_LAYER_DIM, HIDDEN_L

print(f'\nBiLSTM - Fixed Length: {EPOCHS} epochs, Learning Rate: 0.1')
print('=============================================================')
train_model(model_fixed, epochs=EPOCHS, lr=0.1)
print(f'\nBiLSTM - Fixed Length: {EPOCHS} epochs, Learning Rate: 0.05')
print('=============================================================')
train_model(model_fixed, epochs=EPOCHS, lr=0.05)
print(f'\nBiLSTM - Fixed Length: {EPOCHS} epochs, Learning Rate: 0.01')
print('=============================================================')
train_model(model_fixed, epochs=EPOCHS, lr=0.01)
```

```
BiLSTM - Fixed Length: 100 epochs, Learning Rate: 0.1
=============================================================
Epoch 0: training loss 2.421, valid. loss 2.985, valid. accuracy 0.36
0, and valid. RMSE 0.773
Epoch 20: training loss 2.655, valid. loss 2.286, valid. accuracy 0.34
7, and valid. RMSE 0.827
Epoch 40: training loss 2.705, valid. loss 2.568, valid. accuracy 0.36
0, and valid. RMSE 0.867
Epoch 60: training loss 2.429, valid. loss 3.343, valid. accuracy 0.33
3, and valid. RMSE 0.933
Epoch 80: training loss 2.519, valid. loss 2.746, valid. accuracy 0.37
3, and valid. RMSE 0.853
FINAL: training loss 2.295, valid. loss 2.645, valid. accuracy 0.333,
and valid. RMSE 0.853

BiLSTM - Fixed Length: 100 epochs, Learning Rate: 0.05
=============================================================
Epoch 0: training loss 2.181, valid. loss 1.717, valid. accuracy 0.41
3, and valid. RMSE 0.760
Epoch 20: training loss 1.571, valid. loss 1.622, valid. accuracy 0.34
7, and valid. RMSE 0.880
Epoch 40: training loss 1.413, valid. loss 1.762, valid. accuracy 0.41
3, and valid. RMSE 0.813
Epoch 60: training loss 1.585, valid. loss 1.639, valid. accuracy 0.48
0, and valid. RMSE 0.653
Epoch 80: training loss 1.560, valid. loss 1.699, valid. accuracy 0.42
7, and valid. RMSE 0.827
FINAL: training loss 1.561, valid. loss 1.977, valid. accuracy 0.427,
and valid. RMSE 0.800

BiLSTM - Fixed Length: 100 epochs, Learning Rate: 0.01
=============================================================
Epoch 0: training loss 1.293, valid. loss 1.739, valid. accuracy 0.42
7, and valid. RMSE 0.787
Epoch 20: training loss 1.012, valid. loss 1.213, valid. accuracy 0.42
7, and valid. RMSE 0.733
Epoch 40: training loss 1.024, valid. loss 1.373, valid. accuracy 0.36
0, and valid. RMSE 0.840
Epoch 60: training loss 1.034, valid. loss 1.218, valid. accuracy 0.44
0, and valid. RMSE 0.747
Epoch 80: training loss 0.948, valid. loss 1.397, valid. accuracy 0.36
0, and valid. RMSE 0.867
FINAL: training loss 1.001, valid. loss 1.373, valid. accuracy 0.373,
and valid. RMSE 0.813
```

Out[25]:
```
(1.0010928238431613, 1.372769546856483, tensor(0.3733), 0.8133333333333334)
```

BiLSTM - Variable Length Input

We can see the implemented model class in BiLstmClassifier class (BiLstmClassifier.py).

BiLstmVariableLength has the following features:

- Word-Embedding Layer. # Embeddings: Vocabulary Size, Embeddings size: 50
- Bi-directional LSTM Layer. Input size: 50, Hidden size: 60
- Linear Layer. Fully connected layer, Input size: 60, 3 output features (roles)
- Dropout: 0.7
- Variable Length Input. Uses PyTorch's pack_padded_sequence (https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pack_padded_sequence.html) to create sequences of variable length.

In [26]:
```
model = BiLstmVariableLength(vocab_size, EMBEDDED_LAYER_DIM, HIDDEN_LAYE

print(f'\nBiLSTM - Variable Length: {EPOCHS} epochs, Learning Rate: 0.1
print('=============================================================')
train_model(model, epochs=EPOCHS, lr=0.1)
print(f'\nBiLSTM - Variable Length: {EPOCHS} epochs, Learning Rate: 0.05
print('=============================================================')
train_model(model, epochs=EPOCHS, lr=0.05)
print(f'\nBiLSTM - Variable Length: {EPOCHS} epochs, Learning Rate: 0.01
print('=============================================================')
train_model(model, epochs=EPOCHS, lr=0.01)
```

```
BiLSTM - Variable Length: 100 epochs, Learning Rate: 0.1
=============================================================
Epoch 0: training loss 2.369, valid. loss 2.374, valid. accuracy 0.45
3, and valid. RMSE 0.733
Epoch 20: training loss 2.643, valid. loss 2.622, valid. accuracy 0.30
7, and valid. RMSE 0.973
Epoch 40: training loss 2.428, valid. loss 2.488, valid. accuracy 0.40
0, and valid. RMSE 0.880
Epoch 60: training loss 2.489, valid. loss 2.684, valid. accuracy 0.40
0, and valid. RMSE 0.840
Epoch 80: training loss 2.482, valid. loss 2.574, valid. accuracy 0.40
0, and valid. RMSE 0.827
FINAL: training loss 2.690, valid. loss 2.220, valid. accuracy 0.480,
and valid. RMSE 0.680

BiLSTM - Variable Length: 100 epochs, Learning Rate: 0.05
=============================================================
Epoch 0: training loss 2.134, valid. loss 1.952, valid. accuracy 0.32
0, and valid. RMSE 0.947
Epoch 20: training loss 1.642, valid. loss 1.864, valid. accuracy 0.37
3, and valid. RMSE 0.867
Epoch 40: training loss 1.552, valid. loss 1.858, valid. accuracy 0.38
7, and valid. RMSE 0.827
Epoch 60: training loss 1.526, valid. loss 1.958, valid. accuracy 0.37
3, and valid. RMSE 0.907
Epoch 80: training loss 1.682, valid. loss 2.003, valid. accuracy 0.30
7, and valid. RMSE 0.907
FINAL: training loss 1.642, valid. loss 1.508, valid. accuracy 0.520,
and valid. RMSE 0.653

BiLSTM - Variable Length: 100 epochs, Learning Rate: 0.01
=============================================================
Epoch 0: training loss 1.403, valid. loss 1.383, valid. accuracy 0.50
7, and valid. RMSE 0.653
Epoch 20: training loss 1.063, valid. loss 1.198, valid. accuracy 0.42
7, and valid. RMSE 0.787
Epoch 40: training loss 1.099, valid. loss 1.169, valid. accuracy 0.33
3, and valid. RMSE 0.933
Epoch 60: training loss 1.024, valid. loss 1.307, valid. accuracy 0.40
0, and valid. RMSE 0.813
Epoch 80: training loss 1.058, valid. loss 1.259, valid. accuracy 0.41
3, and valid. RMSE 0.813
FINAL: training loss 1.048, valid. loss 1.352, valid. accuracy 0.387,
and valid. RMSE 0.787
```

Out[26]:
```
(1.0475985146682671, 1.3518649652600288, tensor(0.3867), 0.78666666666
66666)
```

BiLSTM - with pretrained GloVe Word Embeddings

We can see the implemented model class in [BiLstmClassifier class (BiLstmClassifier.py)](BiLstmClassifier.py).

BiLstmGloveVector has the following features:

- Word-Embedding Layer. # Embeddings: Vocabulary Size, Embeddings size: 50
- Bi-directional LSTM Layer. Input size: 50, Hidden size: 60
- Linear Layer. Fully connected layer, Input size: 60, 3 output features (roles)
- Dropout: 0.7
- Uses pretrained GloVe Word Embeddings to initialize weights based on its vocabulary.

In [27]:
```python
def load_glove_vectors():
    """Load the glove Global Vectors for Word Representation"""
    word_vectors = {}

    with open("./glove/glove.6B.50d.txt", encoding="utf8") as f:
        for line in f:
            split = line.split()
            word_vectors[split[0]] = np.array([float(x) for x in split[1
    return word_vectors


def get_embedding_matrix(word_counts, emb_size=50):
    """ Creates embedding matrix from word vectors"""
    vocab_size = len(word_counts) + 2
    vocab_to_idx = {}
    vocab = ["", "UNK"]
    W = np.zeros((vocab_size, emb_size), dtype="float32")
    W[0] = np.zeros(emb_size, dtype='float32')  # adding a vector for pa
    W[1] = np.random.uniform(-0.25, 0.25, emb_size)  # adding a vector f
    vocab_to_idx["UNK"] = 1
    i = 2

    for word in word_counts:
        if word in word_vecs:
            W[i] = word_vecs[word]
        else:
            W[i] = np.random.uniform(-0.25, 0.25, emb_size)
        vocab_to_idx[word] = i
        vocab.append(word)
        i += 1
    return W, np.array(vocab), vocab_to_idx
```

In [28]:
```
word_vecs = load_glove_vectors()
pretrained_weights, vocab, vocab2index = get_embedding_matrix(counts, EM

model = BiLstmGloveVector(vocab_size, EMBEDDED_LAYER_DIM, HIDDEN_LAYER_D

print(f'\nBiLSTM - with pretrained GloVe Word Embeddings: {EPOCHS} epoch
print('=================================================================
train_model(model, epochs=EPOCHS, lr=0.1)
print(f'\nBiLSTM - with pretrained GloVe Word Embeddings: {EPOCHS} epoch
print('=================================================================
train_model(model, epochs=EPOCHS, lr=0.05)
print(f'\nBiLSTM - with pretrained GloVe Word Embeddings: {EPOCHS} epoch
print('=================================================================
train_model(model, epochs=EPOCHS, lr=0.01)
```

```
BiLSTM - with pretrained GloVe Word Embeddings: 100 epochs, Learning
Rate: 0.1
=======================================================================
===============
Epoch 0: training loss 1.766, valid. loss 1.248, valid. accuracy 0.33
3, and valid. RMSE 0.853
Epoch 20: training loss 1.639, valid. loss 1.363, valid. accuracy 0.3
07, and valid. RMSE 0.933
Epoch 40: training loss 1.540, valid. loss 1.507, valid. accuracy 0.4
00, and valid. RMSE 0.800
Epoch 60: training loss 1.596, valid. loss 1.351, valid. accuracy 0.2
93, and valid. RMSE 0.960
Epoch 80: training loss 1.594, valid. loss 1.497, valid. accuracy 0.3
60, and valid. RMSE 0.947
FINAL: training loss 1.476, valid. loss 1.329, valid. accuracy 0.360,
and valid. RMSE 0.907

BiLSTM - with pretrained GloVe Word Embeddings: 100 epochs, Learning
Rate: 0.05
=======================================================================
===============
Epoch 0: training loss 1.487, valid. loss 1.363, valid. accuracy 0.29
3, and valid. RMSE 1.000
Epoch 20: training loss 1.170, valid. loss 1.174, valid. accuracy 0.3
87, and valid. RMSE 0.867
Epoch 40: training loss 1.139, valid. loss 1.152, valid. accuracy 0.4
13, and valid. RMSE 0.640
Epoch 60: training loss 1.035, valid. loss 1.311, valid. accuracy 0.3
07, and valid. RMSE 0.827
Epoch 80: training loss 1.247, valid. loss 1.186, valid. accuracy 0.3
87, and valid. RMSE 0.867
FINAL: training loss 1.191, valid. loss 1.255, valid. accuracy 0.333,
and valid. RMSE 0.973

BiLSTM - with pretrained GloVe Word Embeddings: 100 epochs, Learning
Rate: 0.01
=======================================================================
===============
Epoch 0: training loss 1.042, valid. loss 1.220, valid. accuracy 0.36
0, and valid. RMSE 0.707
```

```
Epoch 20: training loss 1.023, valid. loss 1.171, valid. accuracy 0.3
73, and valid. RMSE 0.920
Epoch 40: training loss 0.970, valid. loss 1.149, valid. accuracy 0.3
87, and valid. RMSE 0.907
Epoch 60: training loss 0.959, valid. loss 1.129, valid. accuracy 0.3
73, and valid. RMSE 0.907
Epoch 80: training loss 0.952, valid. loss 1.149, valid. accuracy 0.3
33, and valid. RMSE 0.987
FINAL: training loss 0.902, valid. loss 1.135, valid. accuracy 0.360,
and valid. RMSE 0.947
```

Out[28]:

```
(0.902417093258329, 1.1352980141410565, tensor(0.3600), 0.946666666666
6667)
```

Testing Several Files

In [29]:
```
@torch.no_grad()
def get_all_preds(model, loader):
    all_preds = torch.tensor([])
    for batch in loader:
        images, labels = batch

        preds = model(images)
        all_preds = torch.cat(
            (all_preds, preds)
            ,dim=0
        )
    return all_preds

def get_num_correct(preds, labels):
    return preds.argmax(dim=1).eq(labels).sum().item()

file_size = [150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 645]
accuracy_fixed = []
accuracy_variable = []
accuracy_glove = []

for i in file_size:
    BATCH_SIZE = int(i * 0.5)
    file_name = f'output/balanced_{i}.csv'
    roles = pd.read_csv(file_name)
    mapping = {'Student': 0, 'Co-Facilitator': 1, 'Facilitator': 2}
    roles['Role'] = roles['Role'].apply(lambda x: mapping[x])
    counts = Counter()

    for index, row in roles.iterrows():
        counts.update(tokenize(row['Text']))

    for word in list(counts):
        if counts[word] < 2:
            del counts[word]

    vocab2index = {'': 0, 'UNK': 1}
    words = ['', 'UNK']
    for word in counts:
        vocab2index[word] = len(words)
        words.append(word)

    roles['Vectorized'] = roles['Text'].apply(lambda x: np.array(encode_

    X = list(roles['Vectorized'])
    y = list(roles['Role'])
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_siz

    training_ds = RolesDataset(X_train, y_train)
    validation_ds = RolesDataset(X_valid, y_valid)

    vocab_size = len(words)
    training_dl = DataLoader(training_ds, batch_size=BATCH_SIZE, shuffle
    validation_dl = DataLoader(validation_ds, batch_size=BATCH_SIZE)

    print('\n*********************************************')
```

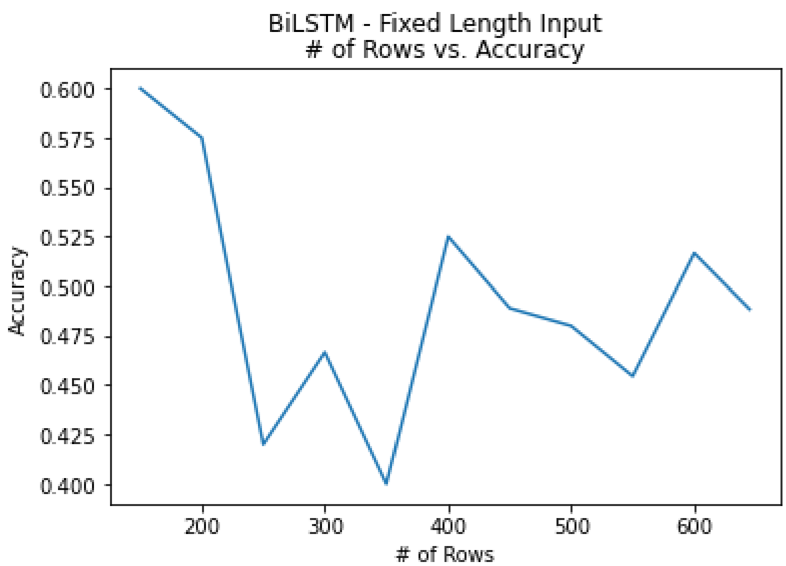
```
    print(f'* Processing file: {file_name} *')
    print('*********************************************')

    print(f'\nBiLSTM - Fixed Length Input')
    print('===========================')
    model_fixed = BiLstmFixedLength(vocab_size, EMBEDDED_LAYER_DIM, HIDD
    train_model(model_fixed, epochs=EPOCHS, lr=0.1, verbose=False)
    train_model(model_fixed, epochs=EPOCHS, lr=0.05, verbose=False)
    _, _, validation_accuracy, _ = train_model(model_fixed, epochs=EPOCH
    accuracy_fixed.append(validation_accuracy)

    print(f'\nBiLSTM - Variable Length Input')
    print('==============================')
    model_varaiable = BiLstmVariableLength(vocab_size, EMBEDDED_LAYER_DI
    train_model(model_varaiable, epochs=EPOCHS, lr=0.1, verbose=False)
    train_model(model_varaiable, epochs=EPOCHS, lr=0.05, verbose=False)
    _, _, validation_accuracy, _ = train_model(model_varaiable, epochs=E
    accuracy_variable.append(validation_accuracy)

    print(f'\nBiLSTM - with pretrained GloVe Word Embeddings')
    print('==============================================')
    word_vecs = load_glove_vectors()
    pretrained_weights, vocab, vocab2index = get_embedding_matrix(counts
    model = BiLstmGloveVector(vocab_size, EMBEDDED_LAYER_DIM, HIDDEN_LAY
    train_model(model, epochs=EPOCHS, lr=0.1, verbose=False)
    train_model(model, epochs=EPOCHS, lr=0.05, verbose=False)
    _, _, validation_accuracy, _ = train_model(model, epochs=EPOCHS, lr=
    accuracy_glove.append(validation_accuracy)
```

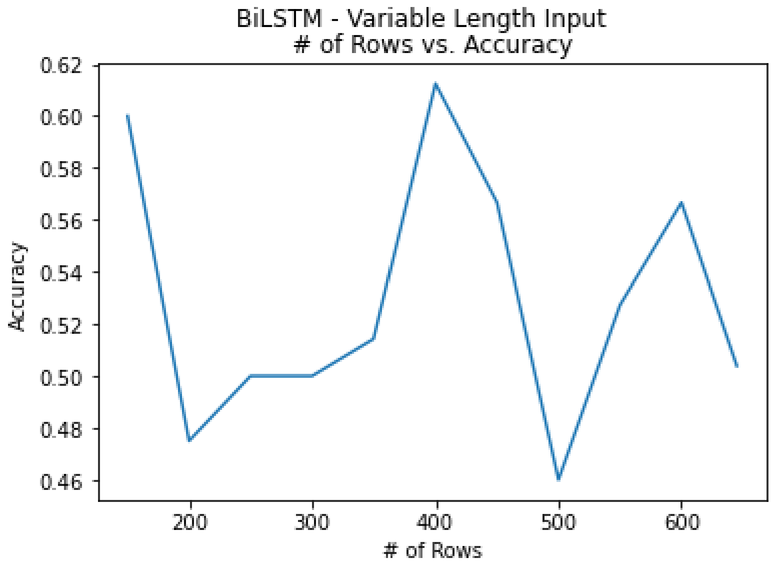
```
********************************************
* Processing file: output/balanced_150.csv *
********************************************

BiLSTM - Fixed Length Input
===========================

<ipython-input-29-919d3a474c88>:43: VisibleDeprecationWarning: Creati
ng an ndarray from ragged nested sequences (which is a list-or-tuple 
of lists-or-tuples-or ndarrays with different lengths or shapes) is d
eprecated. If you meant to do this, you must specify 'dtype=object' w
hen creating the ndarray
  roles['Vectorized'] = roles['Text'].apply(lambda x: np.array(encode
_sentence(x, vocab2index)))
```

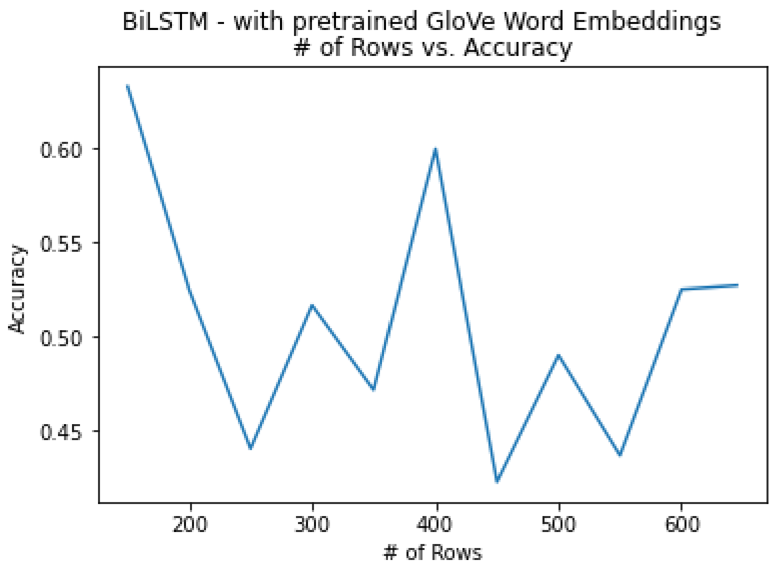
Graphical Performance Analysis

In the following plots we can see the how the model behaves when it is trained with different amounts of data.

In [30]:
```
%matplotlib inline

import matplotlib.pyplot as plt

plt.plot(file_size, accuracy_fixed)
plt.title('# of Rows vs. Accuracy')
plt.suptitle('BiLSTM - Fixed Length Input')
plt.xlabel('# of Rows')
plt.ylabel('Accuracy')
plt.show()

plt.plot(file_size, accuracy_variable)
plt.title('# of Rows vs. Accuracy')
plt.suptitle('BiLSTM - Variable Length Input')
plt.xlabel('# of Rows')
plt.ylabel('Accuracy')
plt.show()

plt.plot(file_size, accuracy_glove)
plt.title('# of Rows vs. Accuracy')
plt.suptitle('BiLSTM - with pretrained GloVe Word Embeddings')
plt.xlabel('# of Rows')
plt.ylabel('Accuracy')
plt.show()
```







Conclusions

- The model with the best performance is **BiLSTM - with pretrained GloVe Word Embeddings**, as we can see that it has a most stable performance as the model is trained with more data.
- The model reaches an approximated Accuracy of 60% for the selected model, and it will probably improve when it is trained with a bigger dataset.
- It is important to see that the first iteration with the smallest dataset with 150 rows reaches a high accuracy. This is happening because over-fitting caused by too few rows, but the accuracy starts to show a more real behavior with bigger datasets.

In [ ]:

```
```