Files in the server (Deep Learning Part)

1. **data_manipulation.py**
    - *SequentialModel_Driver(model_name, hyperparameters, train_S1, train_S2, train_y, cv_S1, cv_S2, cv_y, test_S1, test_S2, test_y, model_saveloc, test_result_loc, S1_all, S2_all, plt_saveloc, is_padded = False, embedding_w=None, is_trainable=False):* This is the function that contains all the necessary code to automate the process of training the model. It takes model name, data and hyperparameters and saves the learning plots and the results of the model
    - *test_model(model, hyperparameters, test_S1, test_S2, test_y, result_loc, device, S1_all, S2_all, is_padded):* This takes a model and tests it on the test data.

2. **deep_models_without_pad.py, deep_models.py**
    They have the following classes (the deep_model_without_pad.py contains the code used in the project). Padding is the case where inbuilt libraries were used (but it takes longer, instead one can make the sentences of the same length by padding 0's before the sentence starts)
    - **RNN**
    - **GRU**
    - **LSTM**

    Each of these classes have a constructor to create an object, overrides *forward* function of the *torch.nn.Model* class and initializes weights based on *'Xavier'* initialization (in the *initialize_weights()* function).

3. **Hyperparameter_tuning.py**
    - *GPU_Queue(object):* This synchronizes between all possible GPUs, and gives job to an idle GPU.
    - *Objective(object) :*

This class is the alternative to the 'optimize_hyperparameters' function. Since we want the tuning of hyperparameters to be divided amongst multiple GPUs. For this 'optuna' need as callable class.

- *hyperparameter_tuner(vocab_size, train_S1, train_S2, train_y, cv_S1, cv_S2, cv_y, model_name, hyp_saveloc):* This calls the functions in the file through an objective object.

- *SequentialModel_hp_tuner(model_name, model, optimizer, train_S1, train_S2, train_y, cv_S1, cv_S2, cv_y, hyp_sugg, hyp_non_opt, device_id):* This is similar to the sequential models of the data_manipulation.py file and returns cross validation accuracy to its caller so that it can optimize over it.

4. **applicationNLI.py**

   This is a class that uses a model to build an application that takes in sentence and infers the relationship between them.

Functions in Colab (TF-IDF features and Logistic regression)

- getDataSet_from_jsonl(dataloc): Reads the json files
- getVocab(train_dict): Gets the vocabulary of distinct words in the train set. (all other words are mapped to 0)
- getVocab_for_db(dataloc_train, dataloc_test, vocab_req = True): Returns the data as a dictionary along with vocabulary (if required)
- basic_preprocessing(s, stopwords, stemmer): This would remove stop words, punctuations and do stemming.
- get_train_TF_IDF_feat(data_dict, vocabulary): Compute TFIDF features of a database.
- get_test_TFIDF_projection(data_dict, tfidf_converter): Compute Tfidf features for cv/test set.

- get_train_test_cv(dataloc_train, dataloc_test, data_for_logReg = True): This combines all the code and returns the data ready for use in training the logistic regression.
- train_softmax_regression(X_train, y_train, X_cv, y_cv, l2_reg, optimizer='lbfgs', max_iter=300): trains the multiclass logistic regression on the data
- def tune_hyperparameters_lg_reg(X_train, y_train, X_cv, y_cv): tune hyperparameters
- get_one_hot_Enc(data_dict, vocab_dict): This is used to make a dataset containing only the indices of the encodings. (This can be directly used to train deep neural networks).
- tokenize_sentences(dataloc_train, dataloc_test): This integrates all the code to generate the data to train the neural networks.