Identifying Well-formed Questions using Deep Learning

by

Navnoor Singh Chhina

A Masters Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering

in the Department of Electrical Engineering
University of Victoria

© Navnoor Singh Chhina, 2020
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
Identifying Well-formed Questions using Deep Learning

by

Navnoor Singh Chhina

Supervisory Committee

Dr. Tao Lu, Supervisor
(Department of Electrical Engineering)
Deep Learning is dominant in the field of Natural language processing, thanks to its performance over the statistical methods. The high performance is driven by the recent transfer learning approaches, where we pre-train a language model over a large corpus and use the pre-trained model for fine-tuning on any specific task. Recent advancements in transfer learning suggest how using the pre-trained model and fine-tuning for a few training steps can give us state-of-the-art results. Therefore, in this project for the classification of well-formed natural language questions, we use both the learning of a model from scratch and transfer learning. We can use the pre-trained base models from BERT, ALBERT and XLNet and add a simple classifier layer, which gives us better results than learning the model from scratch in very few epochs. We also sample a subset of classified queries by our model and run the queries on the Google search engine and confirm that using a model that can identify the well-formedness of the queries would be helpful to the search engine by reducing the downstream compounding errors for the natural language processing pipeline.
Contents

Supervisory Committee ii
Abstract iii
Table of Contents iv
List of Tables vi
List of Figures vii
Acknowledgements ix
Dedication x

1 Introduction 1
1.1 Background .............................................. 3
1.2 Feature Engineering by Faruqui and Das [1] ............... 4
1.3 Key contributions ......................................... 5
1.4 Overview ................................................ 6

2 Related Work 7
2.1 Literature review for latest advancements in Deep learning for NLU . 7
2.2 Dataset for Identifying Well formed queries .................. 19
  2.2.1 Data Pre-Processing for Sequential Models .............. 21
  2.2.2 Data Pre-processing for Transformer Based Models ...... 21

3 Experiments: Model Building 23
3.1 Feature Extraction ......................................... 23
3.2 Experiments: Building Models ............................. 26
  3.2.1 Experiment Results: Model Accuracy on Test set ....... 26
3.2.2 biLSTM Model ........................................... 27
3.2.3 biLSTM with FastText embeddings ....................... 29
3.2.4 biLSTM with Skipgram embeddings ........................ 31
3.2.5 biLSTM with Glove embeddings and subword information for OOV words ........................................ 33
3.2.6 biLSTM with FastText embeddings and subword information for OOV words ........................................ 34
3.3 Transformer Based Models ....................................... 35
  3.3.1 BERT .......................................................... 37
  3.3.2 ALBERT ...................................................... 39
  3.3.3 XLNet ....................................................... 40

4 Model Error Analysis .................................................. 41
  4.1 BERT-Base (Uncased) Model Evaluation ....................... 41
    4.1.1 Correct Predictions: Well Formed Queries ................ 41
    4.1.2 Correct Predictions: Non Well Formed Queries ............ 42
    4.1.3 Incorrect Predictions ..................................... 43
    4.1.4 Incorrect Predictions: Non-well-formed queries predicted well-formed ........................................ 45

5 Conclusions ........................................................... 48

Bibliography ............................................................ 50
List of Tables

Table 2.1 Labels Distribution in Training, Dev and Test Set ............... 19
Table 3.1 Latest Published results ........................................... 26
Table 3.2 Performance comparison using different models on the test set . 26
Table 3.3 Classification Report for GloVe embeddings ...................... 29
Table 3.4 Classification Report for FastText embeddings ................. 30
Table 3.5 Classification Report for Skipgram model ....................... 33
Table 3.6 Classification Results using GloVe embeddings and Subword in-
formation ................................................................. 33
Table 3.7 Classification Report using FastText embeddings and Subword
information ............................................................... 35
List of Figures

Figure 1.1 Source: Faruqui and Das [1] Feed forward network for identifying well formed queries ........................................ 5

Figure 2.1 a) Scaled-Dot Product Attention b) Multi-Head Attention [2] . 9
Figure 2.2 Source: Transformer Architecture [2] ........................... 10
Figure 2.3 Source: BERT [3] .................................................. 12
Figure 2.4 LSTM Cell using Peephole connections [4] ................. 14
Figure 2.5 Hybrid LSTM [5] using word level features and character level features for OOV ........................................ 16
Figure 2.6 Source: Kim et al. [6] ............................................ 17
Figure 2.7 Samples of well-formed and non well-formed queries according to the annotation guidelines. Source: Faruqui and Das [1] ....... 19
Figure 2.8 Samples of human annotations on query well-formedness. Source: Faruqui and Das [1] ........................................ 20
Figure 2.9 10 Samples from the training data after transforming labels . 20

Figure 3.1 biLSTM Model Architecture ................................... 28
Figure 3.2 biLSTM-Dense Model Architecture ........................... 28
Figure 3.3 Accuracy using Glove embeddings ............................ 29
Figure 3.4 Loss using GloVe embeddings ................................. 29
Figure 3.5 Accuracy using FastText embeddings ........................ 31
Figure 3.6 Loss using FastText embeddings ............................. 31
Figure 3.7 Skipgram model architecture, Source: ?] .................. 32
Figure 3.8 Accuracy using Skipgram embeddings ........................ 33
Figure 3.9 Loss using Skipgram embeddings ........................... 33
Figure 3.10 Accuracy using GloVe embeddings and Subword information . 34
Figure 3.11 Loss using GloVe embeddings and Subword information . 34
Figure 3.12 Accuracy using FastText embeddings and Subword information 34
Figure 3.13 Loss using FastText embeddings and Subword information . 34
Figure 3.14 Transformer Based Model Architecture ........................................ 35
Figure 3.15 Adding [CLS] and [SEP] in BERT. The following sample is from
the Training Dataset ................................................................. 36
Figure 3.16 Training Loss for the BERT-Base Uncased .............................. 38
Figure 3.17 Training Loss for the ALBERT-Base ................................. 39
Figure 3.18 Sample of SentencePiece tokenized output used in ALBERT ...... 40

Figure 4.1 Samples extracted from the test data that are correctly classified
using BERT-Base (Uncased) model .............................................. 42
Figure 4.2 Well-formed query sample: California state ............................ 42
Figure 4.3 Well-formed query sample: Stephen Hawkinns .......................... 43
Figure 4.4 Samples extracted from the test data that are correctly classified
using BERT-Base (Uncased) model .............................................. 43
Figure 4.5 Non-Well-formed query sample: Government. Here the model is
able to identify the spelling error and classify the query as non
well-formed ................................................................. 44
Figure 4.6 Non-Well-formed query sample: Classification System ............... 44
Figure 4.7 Well-formed query samples predicted Non wellformed .............. 45
Figure 4.8 The query in the Figure was annotated as a Well-formed query,
but was classified as a Non Well-formed query. We can also
observe that we have a spelling error in the sentence, also there
is a query correction provided by Google Search engine ................. 45
Figure 4.9 Incorrect Prediction: Reject Hotline .................................... 46
Figure 4.10 Samples from the test set where the model’s predictions are not
correct for the Non-wellformed queries .................................... 46
Figure 4.11 Incorrect Classification: Incorrect spelling not detected in Cadap-
illar ................................................................. 46
Figure 4.12 Incorrect Classification: The model incorrectly classified the name
of the inventor, but the query was able to run fine. To over come
this problem, we would need a large knowledge base which the
model can consult before giving the final predictions ................. 47
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Dr. Tao Lu for the continuous support of my MEng study and related research, for his patience, motivation and immense knowledge. His guidance helped me in all the time of research and writing this report.

Also, I would like to thank my family: my parents and my brother for supporting me spiritually throughout writing this report and my life in general.
DEDICATION

This report is dedicated to all my well-wishers - parents, friends and my mentor.
Chapter 1

Introduction

Natural Language Processing or Natural Language Understanding is part of various fields including Linguistics, Computer Science, Artificial Intelligence. Recent advancements in Deep Learning (sub-field of Machine Learning) [3] [7] has allowed for many Natural Language Processing tasks to achieve new state-of-the-art performance. [3] With the introduction of new language representational model BERT, which stands for Bidirectional Encoder Representations from Transformers, which achieves state-of-the-art performance on 11 Natural Language tasks by just using the pre-trained language model and fine-tuning for various downstream tasks including Question-Answering, MultiNLI (Natural Language Inference), etc. Transformers were introduced by Vaswani et al. [2], which uses both single-head attention and multi-head attention mechanisms which aims to find dependencies between distant positions. [8] Recurrent Neural Networks such as Long Short term memory (LSTM), Gated Recurrent Units (GRU) have successfully been used in various tasks including text classification, information extraction, summarization etc. The study by Yin et al. [8] also demonstrates the use of various other deep learning architectures including Convolutional Neural Networks (CNN), Deep Neural Networks (DNN) along with Recurrent Neural Networks (RNN). CNN's include a kernel, which is mapped on the data to perform convolutions, the architecture also uses various other layers such as Pooling and Convolutional layers. The input sequence $x$ containing $n$ tokens, where each token is represented as a $d$ dimensional vector and thus the input is represented as a feature map of $d \times n$ dimension. Yin et al. [8] discusses tasks including Sentiment Classification, Relation classification, Textual entailment, Question Relation Match, etc. Their work finds that Recurrent Neural Networks perform very well and robust in a wide range of tasks except in the task of sentiment detection and question an-
swering. In their experiments, tuning of various hyper-parameters including many hidden layers and batch size significantly varied the performance of the deep learning models, both in case of CNNs and RNNs. While there are numerous tasks in the field of Natural Language Understanding that are out of the scope of this report, the main focus of the project would be one of the most common and important tasks in the NLU field, that is Text/ Sequence Classification. In general, there are three most common types of Machine learning Algorithms:

1. Supervised Learning
2. Unsupervised Learning
3. Semi-Supervised Learning

Text classification falls under the category of supervised learning techniques, where we have labelled data for each sample in our data-set. The labelled data can be manually annotated or the process of labelling the data can be automated. Unsupervised learning algorithms are mainly used to find patterns in the data without labels. Statistical Classification [9] is the problem of identifying the set of categories a new observation belongs on a basis of a training set of data containing observations (or samples) whose category membership is known. For example, assigning a new email sample as “Spam” or “Ham”.

Throughout this project, we derive heavily from pre-trained models and pre-trained static word embeddings, which are initially trained on a large corpus of unlabelled data. For example, the GloVe word vectors that are used in this project were pre-trained on the Wikipedia data and have 6 billion unique tokens. In this project report, we focus mainly on one of the most widely used applications of Natural Language Processing, text classification and a new task introduced by Faruqui and Das [1] in the paper Identifying Well-formed Natural Language Questions. In the first part of the report, we would discuss the dataset released by them with the paper and various feature engineering techniques used by them, mostly that are handcrafted features. We would also discuss the deep learning model used in the paper and their results. Young et al. [10] discusses various recent advancements in Natural Language processing tasks and how it differs from the traditional approach to computation linguistics. Their paper also discusses how handcrafted features are very time consuming to create and still are often incomplete.

In the other part of the report, we would discuss the approach taken by us and compare the results.
1.1 Background

The task of understanding the search queries issued by users is difficult and widely used every day for web browsing for various purposes and applications. Therefore, having a good understanding of well-formedness of the query is of great importance. The introduction of this task can improve the existing NLP engines as they would be aware of how well the query is formed. It would also benefit the end users to have a pleasant experience browsing the web and getting the information they need. The task of identifying well-formed queries is relatively new but derives motivation from various sub-fields of NLP, including question-answering, text summarizing, text classification, etc.

Research by Bergsma and Wang [11] shows that for information to be retrieved from the knowledge bank, the search engine needs to perform query segmentation, by first returning the pages that contain the exact query segments, or using the segmented units if there is no match for the whole query.

Therefore, the task of identifying whether the query is well-formed would help us to be more certain that we can retrieve the most relevant web page as the query requests. Hence, helping to reduce the overall error rate in the information extraction pipeline for search engine.

The task of re-writing a query from an input query by user, to maximizing the chances of getting the most relevant document and document count was also explored using reinforcement learning by Nogueira and Cho [12]. The authors evaluate their approach on three different datasets against strong baselines and show a relative recall score improvement of 5-20 percent.

The data annotated by Faruqui and Das [1] used in this project, define a query to be well-formed if it qualifies the following criteria:
1. The query is grammatical.
2. The query is an explicit question.
3. The sentence does not contain spelling errors.

The scores of the five annotators is averaged. If 4 out of 5 annotators mark the query as well-formed, then they label it as a well-formed query, otherwise the query is non-well-formed.
1.2 Feature Engineering by Faruqui and Das [1]

In Natural Language Processing, the statistics of co-occurrence of various tokens (words or characters) can be used to extract useful information about our data. A *gram* denotes one occurrence of a token (word or character), and we can thus describe tokens as a sequence of *grams*. Therefore, to calculate the statistics for co-occurrence of *grams* we consider $n$ grams at a time, where $n$ can have any value from 1 to $n$. For example, if we have a sequence of tokens and want to calculate $n$-grams for the words and characters in the sequence, where $n = 2$ and 3 respectively, we can describe that as shown in the equations below when the sequence is *How do you use*. We use "_" to donate the white space and treat it as one gram in the example below.

$$Word(2-grams) = (How, doyou), (doyou, use) \quad (1.1)$$

$$Character(3-grams) = (How, ow_), (ow_, w_d), (w_d, do), (do, doy), (doy, oyo), (oyo, you), (you, ou_), (ou_, uu_), (uu_, us), (us, use) \quad (1.2)$$

As we can see in the example equation 1.1 and 1.2, when we use word and character gram information, these features can be used effectively to identify a spelling error. So, Faruqui and Das [1] extracts word-1,2 and character 3,4 grams from all the queries that compose their lexical features. To identify any anomalies in the structure of the query, they also extract syntactic features from each queries using a POS tagger. They also show that using dependency labels and pre-trained word vectors did not improve model performance. They extract POS-1,2,3 gram features using the POS tagger.

Then they sum together all the $n$-gram features of every feature type and concatenate the summed features from word n-grams, character n-grams and POS n-grams, which forms their input layer to the feed-forward network as shown in Figure 1.1.

The model is trained using cross-entropy loss for 50000 training steps. The hyperparameters are tuned to maximize the accuracy on the validation set. The classifier that classifies whether the query is a question is 54.9%. To obtain the word bilstm baseline, they use a single layer encoder of length 50 and get 65.8% accuracy. The results for the concatenated features and passing to a feed word network are in Table
Figure 1.1: Source: Faruqui and Das [1] Feed forward network for identifying well-formed queries

3.1.

1.3 Key contributions

In summary, the key contributions of this report are as follows.

- **Transformer Based model to identify well-formedness of Natural language Question for an end-to-end transfer learning method:** We propose a transformer-based model that is pre-trained on a large corpus and fine-tuned on the dataset constructed by Faruqui and Das [1]. For fine-tuning, we use a **Dense layer** on top of the Transformer based models (BERT, ALBERT, XLNet) with 2 hidden units for our sequence classification task. During experimentation, we notice huge performance gains using a transformer-based model over other transfer learning techniques such as Inductive Transfer Learning proposed by Syed et al. [13], as we use an attention based model as compared to sequential model in Inductive Transfer learning.

- **Pre-trained word vector with Subword based models:** Using pre-trained static word vectors such as GloVe, FastText for the words that are part of the vocabulary and a second model that predicts the OOV words by using subword information performed better than the best model proposed by Faruqui and Das [1], as discussed in the previous section.
1.4 Overview

Chapter 1 gives a brief introduction about the need for identifying well-formed natural questions dataset and the data pre-processing and feature engineering performed by Faruqui and Das [1]. We also discuss the key contributions of this project.

Chapter 2 gives a brief overview of the latest research work done in the field of Natural Language Processing. We also discuss various data pre-processing steps taken by Faruqui and Das [1] and Syed et al. [13] in addition to the pre-processing performed in our experiments.

Chapter 3 discusses the various experiments performed, where we experiment with different features and models. We also compute the accuracy score by using different models on the test set.

Chapter 4 discusses the best performing model’s error analysis on the samples from the test set using the most popular search engine (Google Search).

Chapter 5 provides some final remarks.
Chapter 2

Related Work

2.1 Literature review for latest advancements in Deep learning for NLU

Identifying well formed natural language questions by Faruqui and Das [1]  The task of identifying well-formed queries was first introduced in [1] by Manaal Faruqui and Dipanjan Das. The authors discuss that how understanding queries is a hard task as it involves dealing with ”word salad” text ubiquitously, that is introduced by the users. The authors also discuss how the identification of a well-formed query can help reduce the downstream compounding errors. They also construct a 25,100 query dataset, discussed in Section 1, which is made publicly available. The dataset includes a training, development, and testing set. The distribution of the dataset would be discussed in later section. The dataset can be downloaded at http://goo.gl/language/query-wellformedness. The authors provide samples from the dataset to demonstrate the data preparation procedure.

The authors use hand crafted features such as word n-grmas, character n-grams and POS n-grams to represent the queries. Specifically, they extract character 3,4-grams, word 1,2-grams and POS 1,2,3-grams using SyntaxNet POS tagger [14]. All the features are represented as a real valued embedding. All the n-grams embeddings of every feature type are summed together and concatenated to form the input layer. They use character n-gram embeddings of length 16, and other features length as 25. They project all the features on a dimension of 128 units and then to a dense layer with 64 units. Batch size was set to 32, learning rate in the range of [0.001 - 0.3] and 50,000 training steps/ epochs. They report a maximum accuracy of 70.7%
when using word-1,2 gram and POS-1,2,3 grams. Faruqui and Das [1] also release
baseline accuracy for various models including word biLSTM, majority class baseline
and question word baseline. For training a word biLSTM encoder model they use
hidden length of 50 units to encode the question and then use this representation in
softmax layer to get the predictions for the label. The model achieves 65.8% accuracy
on the test set.

Inductive Transfer Learning for Detection of Well-Formed Natural Lan-
guage Search Queries  Syed et al. [13] worked on the same problem for identifying
well-formed queries using the dataset released by Google AI: http://goo.gl/language/query-
wellformedness. They show how their model trained on the 25,100 queries gives an
accuracy score of 75.03% and thus increasing the accuracy by 4.33% from the state-
of-the-art accuracy by Google AI [1].

In their model approach, they use Inductive Transfer learning, where they pre-
train their Language Model on WikiText-103, which has 103 million unique tokens.
They use Averaged-SGD Weight-Dropped Long Short Term Memory networks (AWD-
LSTM) [15] as their Language model. Then they fine-tune their language model on
the specific distribution similar to the task they want to achieve. Then they further
fine-tune the weights obtained from fine-tuning on a domain-specific dataset. They
use Gradual Unfreezing, where all the layers are not fine-tuned at the same time.
They unfreeze a layer after every epoch and until all the layers are fine-tuned and
they reach convergence. In the paper [13], they also publish results for other exper-
iments performed using the same model architecture (Inductive Transfer Learning)
but changing the hyperparameters and without pre-training on the WikiText-103 cor-
pus. Their best performing model achieves an accuracy of 75% on the test dataset for
Identifying well-formed queries, which uses Pre-training on WikiText-103 and fine-
tuning with DFT and STLR and gradual unfreezing. [16] DFT is Discriminative
Fine-Tuning, where the learning rate is different in the three different layers instead
of keeping the same learning rate throughout. [17] STLR, which stands for Slanted
Triangular Learning Rates, where the learning rate is increased at the beginning and
then linearly decreases with the number of samples we train our model on.
**Transformers** [2] Transformer architecture was first published by Vaswani et al. [2], where the author proposed a different architecture than the recurrent model architectures such as RNN (Recurrent Neural Networks), LSTM (Long Short-term memory) cells. The transformer relies entirely on the self-attention mechanism. They also show how the existing models such as RNNs, LSTMs fails to capture the longer dependencies within the sentence. The transformer has an encoder-decoder structure, where the encoder maps the input sequence \( x \), into a continuous representation sequence \( y \). Both Encoder and Decoder can be stacked on top of each other multiple times, described as \( Nx \) in Figure 2.2. As described, the model is mainly composed of Multi-head attention modules and Feed Forward layers. The inputs and outputs are embedded into a \( n \) dimensional space. We also add the positional information to the \( n \)-dimensional vector as we are not using a recurrent mechanism. The architecture uses both scaled-dot product and multi-head attention. In addition to attention layers, the model uses Position-wise-feed-forward networks, where two linear transformations are applied with a relu activation function as described by Vaswani et al. [2] in the equation below:

\[
FNN(x) = \max(0, xW_1 + b_1)W_2 + b_2
\]  

\[ (2.1) \]

![Scaled Dot-Product Attention](image1.png)  
![Multi-Head Attention](image2.png)

**Figure 2.1:** a) Scaled-Dot Product Attention b) Multi-Head Attention [2]

In case of Scaled product attention, the equation 2.2, \( d_k \) is the dimension of queries and keys, while the dimension of values is \( d_v \).
\[ \text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V \] (2.2)
BERT [3] BERT, which stands for Bidirectional Encoder Representations from Transformers, was recently released by Google. The main design difference that separates BERT from other similar encoder representations other than Transformers, such as ELMo [18] is the design choice that lets BERT to pre-train bidirectional representations from both the left and right context of the unlabelled text data. In their paper, they introduce a novel technique named Masked LM (MLM) which allows bidirectional training in models in which it was previously impossible and they show that how by just adding output layer, the model was able to achieve state-of-the-art performance for several tasks including SQuAD [19], MultiNLI [20], etc. BERT considers the context information of all the previous and next tokens in the sequence, making it bi-directional in nature. This however, makes the current token to cheat during training, and would not be able to learn useful information. Therefore Devlin et al. [3] randomly masks tokens and later predict them. According to the paper, they randomly mask 15% of the tokens in the given sentence and predict them to train the language model. Along with the Masked LM, BERT uses Next Sentence Prediction as its training objective, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document. BERT is pre-trained using 4 to 16 Cloud TPUs (Tensor Processing Units) and takes about 4 days. For Fine-tuning, all results on the paper were fine-tuned on a single Cloud TPU, which has 64GB of RAM. It is currently not possible to re-produce most of the BERT-Large results on the paper using a GPU with 12GB - 16GB of RAM, because the maximum batch size that can fit in memory is too small. The fine-tuning examples which use BERT-Base should be able to run on a GPU that has at least 12GB of RAM using the hyperparameters given. Therefore in our project, we would only consider the Base models. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. To pre-train the model, we add two special tokens, [CLS] and [SEP] at the start and the end of each sentence respectively, the output token [CLS] is then transformed to a 2 × 1 dimensional vector to be used as a classification layer, where we calculate the probability distribution of whether the second sentence is Next or not using a Softmax function. This pre-trained model can be further used for fine-tuning on various other tasks including sequence classification, Question-Answering, etc. The BERT model obtains state-of-the-art results on eleven different tasks, improving GLUE score [21] by 7.7% absolute points, SQuAD v2.0 Test F1 by 5.1 absolute points. The authors
also discuss the comparison with other popular models like OpenAI GPT, which is uni-directional transformer-based models, and ELMo, which uses concatenation to have the context of previous and future words.

![Figure 2.3: Source: BERT [3]](image)

As shown in Figure 2.3, Devlin et al. [3] concatenate the Position, Segment and Token embeddings to form the embeddings of the input layer.

BERT is very effective as we can use the pre-trained model and just fine-tune on our specific task to achieve state-of-the-art results. In this project, we perform experiments using BERT pre-trained models, specifically BERT-Base and report the results.

**ALBERT** [22] The ALBERT, short for A Lite Bidirectional Encoder Representations from Transformers was published by Lan et al. [22]. The main architecture of ALBERT is very similar to BERT with a few design changes such as parameter sharing within all the transformer layers, factorized embedding parametrization and new training objective. The BERT’s vocabulary, which uses word pieces (uses WordPiece tokenization) are projected to the vectors of the same size as the hidden layers so that the embedding size is equal to the hidden layer size. The author’s intuition is that as the BERT’s word piece embeddings learn context-independent representations in the hidden layers, therefore the size of the hidden layer should be greater than embedding layer size. With the increase in the size of vocabulary, it becomes computationally expensive to get word vectors. Therefore, the approach followed is to project the embeddings to a lower-dimensional vector space and then project them to the size of the hidden layers. In BERT, all transformers have separate feed-forward neural networks. However, tying the parameters together significantly decreases the number of parameters of the model. In ALBERT, the author uses a new training objective, which is the sentence order prediction, where we classify whether the order
of sentences are correct or not, which is different than the next sentence prediction as used in BERT. The positive examples for sentence order prediction are the two consecutive sentences, X and Y, whereas the negative example is the order of X and Y swapped, i.e. Y and X.

ALBERT’s Base model uses 18 Million parameters as compared to 108 Million parameters in BERT’s Base model, because of sharing of parameters in ALBERT. The downside of parameter sharing is that ALBERT takes much longer time per training step as compared to BERT. We have also experimented with ALBERT-base for the query well-formedness task and reported the results.

**XLNet [23]** XLNet uses an auto-regressive approach to train the language models, that is it predicts each word using the combination of other words in the sequence. Therefore, given a sequence \( x \), the model calculates the probability \( Pr(x_i|x_{<i}) \), where \( i \) is the current token that we want to condition on the previous tokens.

The main difference between the model architectures of the XLNet model than the BERT or ALBERT models is the use of Transformer XL [24] instead of basic Transformers. When dealing with a large sequence of words, it would be infeasible to use them due to memory constraints. All the computations performed by Transformers are stateless and therefore can be parallelized. The Transformer XL adds recurrence to the vanilla Transformer at the segment level by caching the hidden state values from the previous state and sending them to the current step as key and value pairs. In the BERT and ALBERT models architectures, the authors explicitly add a positional embedding layer, and during the masking of 15% random tokens, all the information about the positions of the tokens and other information is masked as well. The main problem during fine-tuning the BERT model is the \([\text{MASK}]\) token never appears when fine-tuning the BERT model, therefore there is a train-test skew. Also, the predictions made by BERT are independent and therefore can be performed in parallel. Thus, it is not able to learn dependencies within its predictions.

**Long Short-term Memory [25]** LSTMs were first introduced by Hochreiter and Schmidhuber [25] in 1997. The idea was to model predictions as a dynamic classifier rather than a static classifier such as a feed-forward neural network. Here we provide the output of the model at a time step to the next time step as an input, therefore we feed the signals back into the network. This property of the LSTMs allows them to look back in time at various time steps. LSTM consists of a memory cell which
maintains its states over time. It regulates the information between different memory units using gating units. It consists of Input gate, output gate and a forget gate. [4] LSTMs solve the problem of vanishing gradients and exploding gradients by the sharing of Weight matrix or parameters in LSTM cells.

![LSTM Cell using Peephole connections][4]

**Figure 2.4: LSTM Cell using Peephole connections**[4]

**Enriching word vectors using Sub word information** [26] Word vectors have a fixed representation learned using unsupervised learning from a large corpus, which can include billions of tokens. The most common way to learn representations of words is using a skip-gram model or continuous bag of word model, which try to learn the representations of words by their context [? ]. Skip-gram model tries to predict the target word given the context words. Target word can be any word in the sequence we want to compute the vector for. Context words are the words surrounding the target word. Continuous Skip-gram model uses subsampling of frequent words that results in significant speedup and also allows the model to learn more regular word representations. CBOW (Continuous Bag of Words) model aims to learn the word representations from the target word. The target word makes predictions about the context words in this case. In order to use subword information, we store all the character n-grams of the word/token $w$ along with the word itself in a bag. For example, a subword representation for a word *there* where $n$ is 3 and the bag of the
subwords represented as \( bg \) is

\[
bg = [\text{<th, the, her, ere, re>}, \text{<there>}] 
\]

Using the subword information, we can get the word vectors for out of vocabulary words as well. We sum the word vectors for individual character grams and get the word vector representation for out of vocabulary (OOV) words. This can be very useful for generating the word vectors that are not in the vocabulary or the training dataset.

We would show later how we use the subword information in our experiments and get better performance than using a vector of zero for OOV words.

**Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models** [5] Luong and Manning [5] proposed a new model architecture to deal with rare or OOV words in the NMT (Neural Machine Translation) systems. Luong and Manning [5] build a hybrid system that translates most words using the word level information but when encounters a rare or OOV word, the model consults the character components. Therefore providing a unique way to deal with the OOV words. The two main advantages of using a hybrid approach are that it is much faster and easier to train the model than the character-based model and it never produces unknown words as in the case of word-based models. As we can observe in the Figure 2.5, we feed the sequential model with the word level information in the encoder and decoder for the Neural Machine Translation task but when we encounter a rare word that is not in the word Vocabulary, we use another character-based sequential model that predicts the rare word and thus we substitute the \(<unk>\) token with the predicted word. In the paper, they train a deep LSTM model over the characters and use the final hidden at the top layer to create an on-the-fly word representation and replace the \(<unk>\) token.
Learning to Generate Word Representations using Subword Information

Kim et al. [6] introduce a way to generate the word representations using subword information from characters by a CNN model to address the problem of OOV words. Unlike previous models that are used to learn word representations from a large corpus, Kim et al. [6] take a set of pre-trained word embeddings and generalize it to word entries, including OOV words. They also show that their proposed model outperforms the baseline models that use words or subword units as their atomic units.

They use a character-based convolutional module to capture the most informative aspects of any task. They use one-hot encoding to quantize the character inputs.
They use the concatenation of these character units to form a word, where each word represented as $r$. The input representation to the convolution module is of the following form, where $n$ is the length of the word and $c$ is the one-hot encoding representation:

$$r = c_1 \oplus c_2 \oplus c_3 \ldots \oplus c_n \quad (2.3)$$

The convolution filter is denoted as $F$ and $h$ is the width of the filter. They perform the convolution operation on the padded representation as in equation below:

$$s_i = \tanh(F \cdot r_{i:i+h-1} + b) \quad (2.4)$$

The vector $s_i$ is of the same size as the input length because of the zero paddings. Thus they use a max-pooling layer to extract the salient features from $s_i$. Therefore each feature derived from the strided max-pooling operation has a summary of subword information in the character sequences. They use the following equation for extracting these salient features:

$$e_i = \max[s_{k:k+(i+1)-1}] \quad (2.5)$$

In this example, they process the word “uncovered” that is included in pre-trained word embeddings. In the convolution module, they use three filters with 5-width, add two zero-paddings and other width filters, and set the width of the stride to 3 for pooling. They use a single highway network and then make this resultant representation similar to the pre-trained word embedding of “uncovered”. After the training converges, we can represent words that are lexically related to the input word
but that are not included in pre-trained word embeddings, like “uncovering”, as these words share similar character sequences.

They also use a highway network module as shown in Figure 2.6, as the model can train on very deep networks very effectively as the network learns to route information using a gating mechanism. The equation they used for the highway network is as follows, ⊙ denotes the matrix multiplication:

$$ y = T \odot H + (1 - T) \odot e $$

(2.6)

Where $H$ is a non-linear transformation on its input $e$. For $H$ and $T$, they use the following equations:

$$ T = \sigma(W_t \cdot e + b_t) $$

(2.7)

$$ H = \tanh(W_h \cdot e + b_h) $$

(2.8)

They set the size of resultant vector as the pre-trained word embedding using a linear transformation as shown in equation below:

$$ w = W \cdot y + b $$

(2.9)

$w$ denotes the resulting vector from their proposed generated word representation (GRW) model. They then perform the optimization using the pretrained word embeddings from word2vec and use squared Euclidean Distance as follows:

$$ E = \sum_{v \in V_w} ||w_v - \hat{w}_v||^2 $$

(2.10)

where $w_v$ is the word embedding generated from GRW model and $\hat{w}_v$ is the pretrained word embedding from word2vec.
2.2 Dataset for Identifying Well formed queries

The dataset is constructed by manually annotating 25,100 queries from Paralex corpus by Faruqui and Das [1]. The dataset is publicly available and includes a training set, validation set and a test set. The samples of queries in each dataset is listed in Table 2.1.

![Figure 2.7](image)

Figure 2.7: Samples of well-formed and non well-formed queries according to the annotation guidelines. Source: Faruqui and Das [1]

The following three files can be downloaded for the dataset of identifying well-formed natural Language questions, which is in tsv (tab-separated values) format:
1. train.tsv
2. dev.tsv
3. test.tsv

Each file consists of two columns, one for the query itself and the other column represents its averaged well-formedness score across the five annotators.

The distribution of labels for training, development and test sets are given in the Table 2.1 below.

<table>
<thead>
<tr>
<th>Data</th>
<th>Well-formed samples</th>
<th>Non-Well-formed samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>6799</td>
<td>10702</td>
</tr>
<tr>
<td>Dev Set</td>
<td>1458</td>
<td>2293</td>
</tr>
<tr>
<td>Test Set</td>
<td>1480</td>
<td>2371</td>
</tr>
</tbody>
</table>

Table 2.1: Labels Distribution in Training, Dev and Test Set
Figure 2.8: Samples of human annotations on query well-formedness. Source: Faruqui and Das [1]

10 randomly selected samples of training data after performing label transformation can be seen in figure 2.9.

<table>
<thead>
<tr>
<th>Query</th>
<th>Wellformedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>7044</td>
<td>How many calories is a piece of lobster or crab? 0</td>
</tr>
<tr>
<td>7323</td>
<td>What are the two peninsulas in Alaska? 0</td>
</tr>
<tr>
<td>5375</td>
<td>How much money does an HVAC technician earn? 1</td>
</tr>
<tr>
<td>10182</td>
<td>How many floors is there in the silver trench? 0</td>
</tr>
<tr>
<td>1225</td>
<td>Who is governor of Wyoming? 0</td>
</tr>
<tr>
<td>16206</td>
<td>What are wild life sanctuaries? 1</td>
</tr>
<tr>
<td>17395</td>
<td>How can you beat mission 5 on clubpenguin? 1</td>
</tr>
<tr>
<td>11095</td>
<td>What is the population of Western Australia? 1</td>
</tr>
<tr>
<td>1332</td>
<td>What is Europe’s whole population? 0</td>
</tr>
<tr>
<td>5714</td>
<td>What is shahid’s email? 0</td>
</tr>
</tbody>
</table>

Figure 2.9: 10 Samples from the training data after transforming labels

As analysed in [1], we consider a query to be well-formed if 4 out of 5 annotators marked the query as well-formed, therefore the queries with the well-formedness probability, $p_{wf} \geq 0.8$ is considered well-formed and score below 0.8 is considered non well-formed (Figure 2.9).
2.2.1 Data Pre-Processing for Sequential Models

As the task of identifying the well-formed questions take both the semantics of the queries and syntactical nature of the query into consideration, therefore we do not remove the stop-words [27]. We perform experiments with both the uncased characters in the sequence and cased sequences characters. We also generate our own cased and uncased word vector representations of words from the training corpus. Apart from getting the word embeddings, we also extract character level embeddings from the training data. We train 2 to 6 character n-grams using skip-gram and Continuous Bag of Words (CBOW) models to get the subword information [28]. Subword information is very useful when we have many Out-of-vocabulary (OOV) words. We use the same technique to train the unsupervised learning model to get subwords as demonstrated by Bojanowski et al. [28], where a vector is associated with all the n-grams and to get the embedding of a new word, which is not in our vocabulary, we sum all the vectors for the characters in the word to get a vector representation, which we use as embedding for any out of vocabulary word. For example, when we take n = 3 in the word “anywhere”, we get the following subwords:

< an, any, nyw, ywh, whe, her, ere, re >

and the token < anywhere >.

Bojanowski et al. [28] discusses how using the subwords can help us get embeddings for OOV (out of vocabulary) words.

We perform experiments using embeddings trained on our training dataset using skip-gram and cbow models as Mikolov et al. [29]. We also performed experiments using pre-trained word embeddings such as Glove [30] and fast-text [31] embeddings, which are pre-trained on billions of tokens.

2.2.2 Data Pre-processing for Transformer Based Models

The data pre-processing steps we perform to modify the data to feed it to the Transformer-based model is different than the steps taken in the previous section, where we use uncased or cased words and build our vocabulary. As we perform experiments with different transformer-based models: BERT, ALBERT and XLNet, the data preprocessing steps for all the three models is very similar with a few changes in the way we insert the special tokens, [CLS] and [SEP]. We use the tokenizers and
vocabulary provided by the pre-trained models for best results. In the case of BERT and ALBERT, we add the special tokens at the start and end of each sentence respectively. While in XLNet, we add the special tokens at the end of each query. We then use the $[CLS]$ token to get the pooled vector representation for a complete sentence, which we use for fine-tuning on our classification task. We then apply attention and segment masks to all the tokens other than the padding tokens, where we mask apply mask of zeros. We also do not freeze the embeddings from any transformer-based model and, therefore, train end-to-end models.
Chapter 3

Experiments: Model Building

In this chapter, we discuss the various experiments performed using different deep learning models and compare our results with the baseline results published by Faruqui and Das [1] and other results published on the same dataset discussed in Chapter 2 by Syed et al. [13].

3.1 Feature Extraction

We perform experiments using different features such as using static and contextualized word embeddings. For static word embeddings, we use Glove [30], fasttext [32], where we perform most experiments using pre-trained word vectors of dimension 300. We also train our word embeddings using character n-grams-2,3,4,5,6. The model we use to train the embeddings from character grams were skip-gram and cbow [29]. The embeddings generated using the skip-gram and cbow models are word vector of dimension 300. The contextualized embeddings are extracted in experiments using BERT, ALBERT, XLNET embeddings. These embeddings are not static and depend entirely on the context, so we might have different word vectors for same words, if the words are in different context. We can use these models to train on a large corpus and then fine-tune on various downstream tasks such as Classification, Question Answering, Machine Translation etc. As BERT, ALBERT and XLNet are pre-trained on a large corpus, the number of parameters is very large as compared to the other sequential model used.

We use pre-trained static word vectors that are either downloaded or trained using unsupervised learning algorithm such as skip-gram on the training data, in the
experiments where we use the bidirectional LSTM model. The word vectors are stored in the form of a python dictionary, that maps a set of objects (“keys”) to another set of objects (“values”). All the words in the vocabulary are stored as keys, while the corresponding word vector as values. We can then use the embedding matrix to get the word vector corresponding to each word in our vocabulary.

While using pre-trained embeddings, each word in our vocabulary is assigned an ID. In sequential models, our vocabulary consists of all the words in the GloVe’s dictionary. We can include or omit punctuation or stopwords here. Stopwords are the words which occur in high frequency throughout our data or many datasets. We may omit the stopwords for our model to learn more useful information and not focus on the common words. In this project, we include all stopwords and punctuation in our vocabulary as our task is to identify the well-formedness of the queries both semantically and syntactically. After we build the vocabulary, we can use the downloaded static embeddings such as Glove or Fasttext to form the embedding matrix. Once we have the word vectors corresponding to each word in our vocabulary, it is then easy for us to build an embedding matrix or a lookup table which represents all the vectors corresponding to each ID.

The main issue we encounter is with the Out Of Vocabulary (OOV) words. Out of Vocabulary words that are often represented in short as OOV, are the words that we encounter during the development or testing dataset that we did not see during training time. This becomes hard to learn when we build our embeddings using techniques discussed above. In the case where we use pre-trained word embeddings such as Glove or Fasttext, we can use all the vectors present in our pre-trained model, even during testing. If the word is not present in the testing set and the pre-trained embeddings, we usually represent the word with a vector of zeros.

In case, where we train our word embeddings using the training data, if the tokens in development set or testing set are out of the vocabulary, then we replace them by the token $\texttt{< unk >}$, which we map to a vector of zeros, same as the dimension as the other word vectors. Therefore, it might be more difficult to get better performance if most of the tokens during the testing time are OOV because that means we would use limited information about the words in the sentence.

However, if we use the subword information of words, we can deal with the problem of representing the OOV words as vectors of zeros. The idea of using subword information for OOV was described by Bojanowski et al. [26]. In this project, we perform experiments where we use subword information of words to generate a vector
or embedding representation for OOV words. We mainly use the skip-gram model (Unsupervised learning algorithm discussed in Section 3.2.4) to train on our training dataset, where we use 2 to 6 character grams. During model evaluation, if we encounter an OOV word, we use the concatenated representation of the subwords to represent the word vector. For example, let’s consider that the word *attention* is an OOV word. For \( n = 3 \), we can represent the word as

\[
<\text{at},\text{att},\text{tte},\text{ten},\text{nti},\text{tio},\text{ion},\text{on}>, <\text{attention}>
\]

For each subword in our example word, if we have some representation learnt during training, we would assign the representation to the subword, otherwise, we would assign a vector of zeros to the subword. Finally, we just concatenate these representations and allocate the concatenated representation to form a word vector of the example word *attention*. Then we sum all the concatenated vectors to make a single vector representation for the OOV word.

Apart from using the concatenated representation of words during prediction time, we also perform experiments with giving each OOV word a random vector from a normal distribution instead of a zero vector, as it has proven to be useful in some scenarios. We do not notice much performance gains using these features, therefore do not include them in the report.
3.2 Experiments: Building Models

3.2.1 Experiment Results: Model Accuracy on Test set

In this section, we would discuss various experiments where we implement and test models. We would discuss the model architectures in detail and then look at their results on the test set. In all the experiments using biLSTM models, we use ADAM optimizer, categorical cross-entropy loss function, 0.001 as learning rate, batch size of 64, 100 epochs and use early stopping, we use validation accuracy as early stopping measure.

Table 3.1: Latest Published results

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy%</th>
</tr>
</thead>
<tbody>
<tr>
<td>word bi-LSTM baseline[1]</td>
<td>65.8</td>
</tr>
<tr>
<td>word-1,2 char-3,4 POS-1,2,3[1]</td>
<td>70.2</td>
</tr>
<tr>
<td>word-1,2 POS-1,2,3 [1]</td>
<td>70.7</td>
</tr>
<tr>
<td>Inductive Transfer Learning [13]</td>
<td>75.0</td>
</tr>
</tbody>
</table>

Table 3.1 shows the results from the Faruqui and Das [1] and Syed et al. [13] papers on test set using the same dataset.

Table 3.2: Performance comparison using different models on the test set

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>word bi-LSTM (Skipgram embeddings)</td>
<td>67.5</td>
</tr>
<tr>
<td>word bi-LSTM (GloVe embeddings)</td>
<td>68.2</td>
</tr>
<tr>
<td>word bi-LSTM (FastText embeddings)</td>
<td>70.4</td>
</tr>
<tr>
<td>bi-LSTM (FastText, Subword Information)</td>
<td>71.4</td>
</tr>
<tr>
<td>biLSTM (GloVe, Subword information)</td>
<td>72.7</td>
</tr>
<tr>
<td>BERT-Base(Uncased)</td>
<td>81.6</td>
</tr>
<tr>
<td>BERT-Base(Cased)</td>
<td>80.0</td>
</tr>
<tr>
<td>ALBERT-Base(Uncased)</td>
<td>78.3</td>
</tr>
<tr>
<td>XlNet-Base (Cased)</td>
<td>77.3</td>
</tr>
</tbody>
</table>
3.2.2 biLSTM Model

The biLSTM model uses a bidirectional mechanism, where we concatenate the hidden representations of the cell outputs of two uni-directional LSTMs. By using the bidirectional model, we hope to learn more context information as we take into account the words before and after a given word rather than just the previous words information. Therefore we can represent the output of the biLSTM at time $t$ as

$$\hat{y}^{<t>} = g(W_y[\vec{a}^{<t>}; \bar{a}^{<t>}] + b_y) \quad (3.1)$$

Here, $\hat{y}^{<t>}$ is the output of the LSTM Cell at time $t$. $W_y$ is our weight matrix and $b_y$ is the bias. $\vec{a}^{<t>}$ represents the hidden state from the uni-directional LSTM cell from left to right at a given time $t$, whereas $\bar{a}^{<t>}$ represents the hidden state from right to left. We obtain the output by concatenating both states. $g$ represents another activation function that can be relu or tanh [33].

In the experiments where we use Bidirectional LSTM, we use a wrapper from Keras. This wrapper takes a recurrent layer in our LSTM as an argument. It also allows us to specify the merge mode, that is how our forward and backwards output should be combined. The default mode when using this wrapper is to concatenate the outputs. We can also add together the outputs using the 'sum' argument, multiply the outputs using 'mul' or average the outputs using 'ave' argument. In this project, we simply concatenate the outputs together. During the analysis of data, we select the sequence length as the maximum length in our training data. We then pad the sequence with [PAD] token and assign it a vector of zeros, same as the dimension of the static word vectors used in this project, i.e. 300-dimensional vectors.

The main underlying theory for finding relations between words in an unsupervised setting is the statistics of the word occurrence. GloVe provides an approach to get information from these statistics by not only the Global features from the word-word co-occurrence matrix but also the local statistics from the corpus. We use 64 units in LSTM Cells, and get an accuracy score of 68.23% on the test set using the GloVe embeddings and model architecture shown in Figure 3.2. GloVe pre-trained embeddings can be downloaded from https://nlp.stanford.edu/projects/glove. In this experiment, we use the word vectors trained on 6 Billion tokens, where the tokens are uncased and the vocabulary size is 400K. We use validation accuracy as an early stopping measure and wait for 5 epochs before stopping the training and use the best-weights checkpoint to test the model on our testing data.
In Figure 3.2, we use a shared Dense layer on top of the biLSTM model with 16 units. From experiments, we almost always got better performance using the biLSTM-Dense model rather than using Basic biLSTM model architecture shown in Figure 3.1, because we would have more information about the individual tokens in bidirectional model as compared to unidirectional. All the results for word bi-LSTM reported here, use the architecture as in figure 3.2.
Figure 3.3 plots the number of epochs on the X axis and accuracy on the y axis. We can notice the performance drop in the accuracy in the validation set after 35 epochs.

Here, precision = $\frac{TP}{TP + FN}$, where TP is True Positive and FN is False Negative.

Recall is $\frac{TP}{TP + FN}$

F1 Score is $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$

Table 3.3: Classification Report for GloVe embeddings

<table>
<thead>
<tr>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Well-formed</td>
<td>69</td>
<td>88</td>
<td>77</td>
</tr>
<tr>
<td>Well-formed</td>
<td>65</td>
<td>37</td>
<td>47</td>
</tr>
</tbody>
</table>

3.2.3 biLSTM with FastText embeddings

In this section, we would discuss the fast-text pre-trained word vectors [34]. Mikolov et al. [34] uses the continuous bag-of-words model as used in word2vec [35] with several improvements for learning richer word vectors. First, they change the objective function, where initially the context words would be used to predict the target word. They add a set of negatively sampled examples, $N_c$ from the vocabulary. The loss function used here is
\[
\log(1 + e^{-s(w, C)}) + \sum_{n \in N_c} \log(1 + c^{s(n, C)})
\] (3.2)

The context is represented by the average of word vectors \(u_w^{'}\) of each word \(w^{'}\) in its window and the scoring function, \(s(w, C)\) is given in the equation below:

\[
s(w, C) = \frac{1}{|C|} \sum_{w^{'} \in C} u_{w^{'}}^T v_w
\] (3.3)

To avoid overfitting of the parameters by considering all the words in the vocabulary, Mikolov et al. [34] sub-samples frequent words with the probability \(p_{\text{disc}}\) to discard the word as follows:

\[
p_{\text{disc}}(w) = 1 - \sqrt{\frac{t}{f_w}}
\] (3.4)

where \(f_w\) is the frequency of the word.

A context vector is formed by simply averaging the word vectors in it. Therefore, there is no information about the relative position of the words. They perform position-dependent weighting and thus the context vector is represented the average context words re-weighted by their position vectors. Then they also add phrase representations and subword information to get richer word vectors. The models trained on the Wikipedia and News corpora, and on the Common Crawl, were published at the https://fasttext.cc/ website.

Here, we experiment with the vec file trained on Common Crawl with 600 Billion tokens and contain 2 million word vectors each of depth 300. The files can be downloaded at https://fasttext.cc/docs/en/english-vectors.html. By using FastText pre-trained word embeddings, we get an accuracy of 70.36%.
3.2.4 biLSTM with Skipgram embeddings

Skip-gram is one of the unsupervised learning techniques used to find the most related words for a given word. We can generate static word representations by training the model on a large corpus and then use the learned representations as pre-trained word vectors as features in other datasets. Skip-gram is used to predict the context word for a given target word. In Figure 3.7, $w(t)$ is the Target word and there is one projection layer, which performs the dot product between the weight matrix and the input vector $w(t)$. There is no activation function used in the projection layer and the result of the dot product is passed to the output layer. The output layer computes the dot product between the output vector of the hidden layer and the weight matrix of the output layer. We use a softmax function to find the probability distribution of the context words.

Training a Skip-gram model is exactly the opposite for training a Continuous Bag of Words (CBOW) model. In CBOW, the model architecture is very similar to that of a Feed-Forward network.

We train embeddings on our training dataset by using a skip-gram algorithm to generate word vectors of length 300. We also use subword information to get the word embedding for the unseen words during prediction. By using these unsupervised word embeddings, we get an accuracy of 67.5% using subword information. As the total number of tokens were 3812, therefore we could not capture the same amount of information as we can using the pre-trained embeddings that are pre-trained on billions of tokens.
Where $w(t)$ represents the context word and $w(t-2), w(t-1), w(t+1), w(t+2)$ represent the target words.
Figure 3.8: Accuracy using Skip-gram embeddings

Figure 3.9: Loss using Skipgram embeddings

Table 3.5: Classification Report for Skipgram model

<table>
<thead>
<tr>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Well-formed</td>
<td>75</td>
<td>71</td>
<td>73</td>
</tr>
<tr>
<td>Well-formed</td>
<td>57</td>
<td>61</td>
<td>59</td>
</tr>
</tbody>
</table>

3.2.5 biLSTM with Glove embeddings and subword information for OOV words

We use the same pre-trained embeddings from GloVe as in Section 3.2.2 and use subword information model, which we train by using skip-gram on the training data as in section 3.2.4. Here we ignore all the embeddings from the training data that are in GloVe. And rather than using a \(<unk>\) token, we assign a unique word vector to each OOV word in our corpus by using the subword information of the OOV word. As we can observe in figure 3.10 that our model performs better than just using the GloVe embeddings.

Table 3.6: Classification Results using GloVe embeddings and Subword information

<table>
<thead>
<tr>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Well-formed</td>
<td>79</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>Well-formed</td>
<td>64</td>
<td>67</td>
<td>65</td>
</tr>
</tbody>
</table>
3.2.6 biLSTM with FastText embeddings and subword information for OOV words

We use the pre-trained embeddings from FastText and use the same subword information model used in 3.2.5, which we train by using skip-gram on the training data. Here we ignore all the embeddings from the training data that we can look up from FastText’s embeddings. We can observe that we get slight improvement using subword information as we observed in 3.2.5 because we are feeding more information about the OOV words. We get an accuracy score of 71.4% on the test data.
Table 3.7: Classification Report using FastText embeddings and Subword information

<table>
<thead>
<tr>
<th>Label</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Well-formed</td>
<td>82</td>
<td>68</td>
<td>75</td>
</tr>
<tr>
<td>Well-formed</td>
<td>60</td>
<td>77</td>
<td>67</td>
</tr>
</tbody>
</table>

3.3 Transformer Based Models

Transformer based model architectures have been very popular recently because of the performance gain achieved using attention mechanism rather than recurrent based models such as RNN or LSTM or GRU. Attention mechanism is to create shortcuts between the context vector and the entire sequence input. In this section, we discuss the model architectures, hyperparameters and results of the models on our test data.

The most popular transformer-based models are BERT, ALBERT, XLNet as discussed in Chapter 2. We performed experiments with Base models of BERT, ALBERT, XLNet and report the results in Section 3.2. We use the pre-trained weights and vocabulary files for each model that was released with its code. For performing experiments, we use Tesla P100-PC-IE-16GB GPU and maximum batch size that fit to our GPU’s memory, i.e. batch size of 256 and learning rate of 2e-5.

Figure 3.14: Transformer Based Model Architecture
For each experiment, first we transform and manipulate the data so we can directly feed to our model. We get the pooled output representation of the entire sentence and the sequence outputs for each token in the sentence. In this project, we only use the pooled output and then project the hidden representation to a dense layer with 2 units, which forms our classifier layer. The position of [CLS] and [SEP] tokens vary and depend upon the model architecture being implemented. Figure 3.15 shows the transformation made to the data by adding [CLS] and [SEP] tokens when using the BERT and ALBERT models. We add the [CLS] token at the beginning of each sentence and [SEP] at the end of every sentence. We perform experiments with both cased and uncased versions of BERT and ALBERT. The tokenization used in BERT is WordPiece.

```
['[CLS]',
 'What',
 'are',
 'Mia',
 'Ham',
 '##ms',
 'accomplish',
 '##ment',
 '?',
 '[SEP]']
```

Figure 3.15: Adding [CLS] and [SEP] in BERT. The following sample is from the Training Dataset

As shown in Figure 3.14, the input to the model is the input sequence, where we add the [CLS] and [SEP] tokens as required, then we tokenize the sequence with either WordPiece or SentencePiece depending upon the model being used. After tokenization, we use three embeddings to represent each token in the sequence:

- **Token Embeddings**: Individual token representation as a 768 dimensional vector.
- **Segment Embedding**: Segment embeddings are used to separate two separate sequence with [SEP] token.
• **Position Embedding**: Used to encode order dependencies into the input representation, that is independent of the word context.

In ALBERT and XLNet models we use SentencePiece tokenizer, which uses Byte-Pair encoding [36] and uni-gram language model [37] where we can train in raw sentences. Here the number of unique tokens is predetermined and white space is considered as a symbol.

### 3.3.1 BERT

As discussed in 2, BERT is pre-trained on a very large corpus and fine-tuning on different datasets. In this project, we focus on the sequence classification task of the query. We tokenize the data using the tokenizing method provided by BERT as follows:

**Text normalization**: Convert all whitespace characters to spaces, and (for the Uncased model) lowercase the input and strip out accent markers. E.g., Who didn’t → who didn’t.

**Punctuation splitting**: Split all punctuation characters on both sides (i.e., add white-space around all punctuation characters). Punctuation characters are defined as (a) Anything with a P* Unicode class, (b) any non-letter/number/space ASCII character (e.g., characters like $ which are technically not punctuation). E.g., who didn’t → who didn’t.

**WordPiece tokenization**: Applied whitespace tokenization to the output of the above procedure, and applied WordPiece tokenization to each token separately. E.g., john johanson’s → john johan ##son ’ s.

We use the pre-trained BERT-Base model and transform the data as discussed to feed to the model. We then take the pooled output at the [CLS] token and project it to the lower-dimensional space of length 2, which we use for classification. The model architecture is shown in Figure 3.14. We perform experiments with cased and uncased versions of BERT models and get better performance using uncased version using the same hyper-parameters. The hyperparameters for base models is in Section 3.3.

In Figure 3.16 we can observe that our training loss converges much better than in the case of biLSTM models in Section 3.2.2.
By just using the pre-trained BERT-base model and using a classifier layer, that classifies whether the query is well formed or not, we are able to achieve approximately 6.7 absolute percentage point improvement on the test set over the ULMFit based architecture mentioned in Section 2.
3.3.2 ALBERT

We use the ALBERT pre-trained initial checkpoints from ALBERT-Base-v1 and add a Classifier layer to fine-tune the model for the Sentence Classification task. The classifier layer we used is the same as used in BERT, where we project the input features (input features has size 1024) at $[CLS]$ linearly to 2, for classifying whether the sentence is well-formed or not. We discard the sentence level feature output from ALBERT. We use Adam optimizer [38] and use the same hyperparameters as used in the BERT model. We experiment using 20 and 50 epochs and report the results for 20 epochs. ALBERT model architecture is very similar to BERT with few design choices by which we can significantly reduce the number of parameters than using BERT. The model architecture is discussed in Chapter 2. We use the same hyperparameters for the ALBERT-Base model as used in the BERT-base model and then vary the learning rate and batch size. We report the results for model performance in the Section 3.2 in Table 3.2. We add the $[CLS]$ and $[SEP]$ tokens at the start of the query and its end respectively as we did in BERT but use SentencePiece tokenizer as shown in Figure 3.18 instead of WordPiece (Figure 3.15).

![Figure 3.17: Training Loss for the ALBERT-Base](image)
Figure 3.18: Sample of SentencePiece tokenized output used in ALBERT

3.3.3 XLNet

XLNet model is explained in the Section 2, here we use the [CLS] and [SEP] tokens as in BERT and ALBERT models, but the tokens are added at the end of queries. We use SentencePiece tokenization, similar to that used in ALBERT. We performed experiments using the same hyperparameters for the Base model as discussed in Section 3.3. We also report the results for the models and show how the transformer-based models outperform the recurrent based models in our experiments.
Chapter 4

Model Error Analysis

In Section 3.2, we discuss various deep learning models we used. We conduct various experiments using different models and list the Accuracy score of the trained model on our testing set in Table 3.2. In this section, we would do a detailed analysis of the model outputs and see the type of errors produced by different models. We use the best performing model, BERT-Base (Uncased) and predict the queries on the test data. We then randomly sample the following queries for model evaluation:

1. Queries that are predicted correctly by the model.
2. Queries that are not predicted correctly by the model.

We also run the query using a Google Search, to validate if the model predictions would be practical in the real world.

4.1 BERT-Base (Uncased) Model Evaluation

Using BERT-Base Uncased model, we achieve an accuracy score of 81.6 %, outperforming the State-of-the-art model (ULMFit), that uses transfer learning similar to BERT. Although BERT is a transformer-based model when compared to the Sequential model (LSTM) used in ULMFit model.

4.1.1 Correct Predictions: Well Formed Queries

In the Figure 4.1, the number associated with each query is the Sample ID. To validate the results practically, whether one of the most popular search engines (Google Search) understand the query and return the results without any errors, I select the top 5 queries from the Samples in Figure 4.1 and run the query in the Google
42

Figure 4.1: Samples extracted from the test data that are correctly classified using BERT-Base (Uncased) model.

Search engine. All the queries ran as expected and the Search engine identified the well-formed queries and was able to give back the search results.

![Search engine result](image)

Figure 4.2: Well-formed query sample: California state

### 4.1.2 Correct Predictions: Non Well Formed Queries

In Figure 4.4, which is a sample of 10 queries drawn where the model is correctly able to classify the query as non-well-formed runs perfectly fine using Google Search engine. Therefore, our model is not only able to identify the spelling mistakes as shown in Figure 4.5 in the input query, but also consider the semantic structure of the query.

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>2833 How do plant parts grow toward light?</td>
</tr>
<tr>
<td>1716 When was California founded as a state?</td>
</tr>
<tr>
<td>3338 How did Bernie Mac die?</td>
</tr>
<tr>
<td>3566 What day in 1942 was Stephen Hawking born?</td>
</tr>
<tr>
<td>3759 How do zebras communicate using sound?</td>
</tr>
<tr>
<td>625 How many carbohydrates are in watermelon?</td>
</tr>
<tr>
<td>1386 How many pictures will a 4GB memory card in a ...</td>
</tr>
<tr>
<td>2683 How much does a civil engineer earn in a month?</td>
</tr>
<tr>
<td>3562 How does the cell recycle ATP and ADP?</td>
</tr>
<tr>
<td>1375 What kind of mice do hummingbirds eat?</td>
</tr>
</tbody>
</table>

Name: query, dtype: object
4.1.3 Incorrect Predictions

In this section, we would look at a few examples of the incorrect classifications by the model and list potential solutions of how we can improve the performance. First, we would take a look at the well-formed queries that were incorrectly classified as in Figure 4.7. Then we would look at a few samples where the model incorrectly classified the well-formed queries as non-well-formed.

In Figure 4.9, we can observe that the model might be considering “reject” as a verb. The other query, \textit{Who are oprah winfrey's siblings ?} had a spelling error, for the word \textit{winfreys}.
Figure 4.5: Non-Well-formed query sample: Government. Here the model is able to identify the spelling error and classify the query as non well-formed.

Figure 4.6: Non-Well-formed query sample: Classification System

A potential solution might be to feed the model Part of Speech tagging information in addition to the embeddings using some dependency parser from libraries such as SpaCy.
Figure 4.7: Well-formed query samples predicted Non wellformed

<table>
<thead>
<tr>
<th>Query</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there any money glitches for Mafia Wars?</td>
<td>Non well-formed</td>
</tr>
<tr>
<td>What days are there festivals in Italy?</td>
<td>Well-formed</td>
</tr>
<tr>
<td>Who are oprah winfrey's siblings?</td>
<td>Well-formed</td>
</tr>
<tr>
<td>Where do veterinarians work other than private...</td>
<td>Non well-formed</td>
</tr>
<tr>
<td>What are the word equations for combustion and...</td>
<td>Well-formed</td>
</tr>
<tr>
<td>How much does lard cost a pound?</td>
<td>Non well-formed</td>
</tr>
<tr>
<td>20 newtons equals how many joules?</td>
<td>Well-formed</td>
</tr>
<tr>
<td>How many cups of pecans are in 2 pounds?</td>
<td>Non well-formed</td>
</tr>
<tr>
<td>What is the phone number to the reject hotline?</td>
<td>Well-formed</td>
</tr>
<tr>
<td>Where are snorunts in pokemon pearl?</td>
<td>Non well-formed</td>
</tr>
</tbody>
</table>

Name: query, dtype: object

Figure 4.8: The query in the Figure was annotated as a Well-formed query, but was classified as a Non Well-formed query. We can also observe that we have a spelling error in the sentence, also there is a query correction provided by Google Search engine.

4.1.4 Incorrect Predictions: Non-well-formed queries predicted well-formed

Figure 4.10 shows 10 random samples where the model predicted the queries to be Well-formed, but were labeled as non-well-formed. We can observe that a few queries seem to be well-formed but are annotated as non-well-formed. For example, the query, Where is arginine found? seems to be syntactically and semantically correct, also gives the desired result after running the query through the Google Search Engine but is annotated as a non-well-formed query.
Figure 4.9: Incorrect Prediction: Reject Hotline

Figure 4.10: Samples from the test set where the model’s predictions are not correct for the Non-wellformed queries

```python
2583   How many calories in a baked chicken nugget?
3295   Where is arginine found?
2220   How many cups is equal to 2 quarts?
1838   How do you find lucky?
1330   What does the proteins in a cell do?
1126   How many electoral votes does new oklahoma have?
949    What did henery bessemer invent?
2437   What do cadapillars eat?
974    What is so important for teenagers to eat meals?
1496   What episode does ash and dawn have a baby?
Name: query, dtype: object
```

Figure 4.11: Incorrect Classification: Incorrect spelling not detected in Cadapillar

In both the incorrectly classified samples shown in Figure 4.11 and 4.12, there is a spelling error that was not detected by the classifier.
Figure 4.12: Incorrect Classification: The model incorrectly classified the name of the inventor, but the query was able to run fine. To overcome this problem, we would need a large knowledge base which the model can consult before giving the final predictions.

A potential solution might be to pre-train the language model on a larger corpus and include more tokens in the vocabulary.
Chapter 5

Conclusions

In this report, we discussed the Query well-formedness dataset in detail and how the identification of a query as being well-formed or non-well-formed can help the overall error reduction in the web search engines pipeline and other tasks such as Question-Answering. We discuss about the dataset released by Faruqui and Das [1] and the feature extraction and model building performed by Faruqui and Das [1] and Syed et al. [13]. Faruqui and Das [1] focus on more hand-crafted features such as character-2,3 grams, word-1,2 grams and POS-1,2,3 grams and use a simple feed-forward neural network, while Syed et al. [13] uses transfer-learning, that includes pre-training a language model (LSTM-based model) on a large corpus and then fine-tuning on the dataset. In this project, we use the publicly released dataset that includes training set, validation set and test set. We perform training of the neural network models using the training set and validate the performance on the validation set. Then we use the training model to predict the sequence classification task on the test set. We build sequential and Transformer based models and compare our results with the baseline and state-of-the-art results reported. We discuss the various pre-processing steps taken by us for both the sequential models and Transformer-based models in Chapter 2. Then we discuss about the models used and various features used in each experiment. The results for the accuracy of models on the test set are reported in Table 3.1. My main contribution in this project is to provide a new method to detect whether the queries are well-formed or non-well-formed using two different approaches, the first approach is using the biLSTM model with pre-trained word vectors or training word-vectors from the training data. We propose biLSTM model as an alternative to the approach by Faruqui and Das [1] and Transformer-based models as an alternative to the transfer learning approach proposed by Syed
et al. [13]. We gain 2 percent performance improvement while using the biLSTM model over the feed-forward network and 6.6 percentage points improvement using the BERT-Base model over using the Inductive Transfer learning method proposed by Syed et al. [13] and 10.9 percentage improvement over word-1,2 POS-1,2,3 model by Faruqui and Das [1] on the test set. We can observe that the other transformer-based models were able to perform significantly better than the sequential based models in all the experiments listed in Table 3.2 without performing any exhaustive hyperparameter tuning search.

In the Chapter 4, we sample few queries and test them on Google Search engine to see if the task of identifying well-formed queries would be useful in a practical environment. We use the predictions made by our best model weights, BERT-Base and notice that the model is able to accurately consider into account both the semantic and syntactic structure of the questions.

We can also observe that we were able to get better performance when we use the Subword information of the $< unk >$ word in addition to the pre-trained word embeddings (GloVe in this case) in comparison to the sequential models used by Faruqui and Das [1].

Although the BERT-Base model can very accurately predict the well-formedness of a natural language question, this can, however be improved by providing additional information like the POS features, which might further improve the model performance. Also, we only used base models for our experiments, future work may include trying larger models which have proved to improve the model performance overall. Also, we performed a very limited hyperparameter search, further performing exhaustive hyperparameter search may give us some improvement. We could also try to pre-train the transformer-based model on more relevant and large corpus and experiment to see if we can get better performance.
Bibliography


