Problem Statement

GFG got more than 500 technical articles per day from various contributors from all over the world (mostly from India). And currently, almost 60 (30 internals & 30 externals) reviewers in various computer science domains are working in GFG Content Team. So, it's a very tedious task for the manager to assign those articles among the reviewers daily, and it consumes a lot of time for the manager. So, we are building such a tool that can process the title of the article in such a way that it can assign that article automatically to the corresponding reviewer and it will save lots of time for the manager. For example, an article named "Variables in C Programming Language" will assign automatically to that reviewer who is reviewing the article in the C Programming domain. So, we are going to use the Natural Language Processing and Classification Algorithm for this task.

Raw Data Parsing

We have used the Beatifulsoup library as a handy html parser to extract all the needed data into a pandas dataframe.

In [1]:
```
import requests
from bs4 import BeautifulSoup
import lxml
```

In [2]:
```
source = requests.get('https://origin.geeksforgeeks.org/wp-admin/edit.php?post
_type=post').text
soup = BeautifulSoup(source, 'lxml')
```

Key Words Detection

This part was pretty straightforward — we used a fast langdetect implementation as language detector and feeded it with titles by default to speed up detection — language detection on texts with 10x larger average length is slower.

In [ ]:
```
t = time.time()
    for i, row in df.iterrows():
        try:
            df['lang'][i] = langdetect.detect(df['title'][i])
        except BaseException as e:
            print(e)
            try:
                df['lang'][i] = langdetect.detect(df['text'][i])
            except BaseException as e1:
                print(e1)
                df['lang'][i] = 'not_defined'

print('exec time lang detection: ', time.time() - t)
```

Text Preprocessing Logic

Text vectorisation logic is one of the core algorithm decisions author had to come up with. We had a large and variative enough corpus of around 1M articles to use pretrained word embeddings instead of the basic TF-IDF approach. But first we had to perform common tokenization and stemming procedures. We have used stopwords list and Porter Stemmer from the nltk library.

In [ ]:
```python
from gensim.parsing import PorterStemmer
global_stemmer = PorterStemmer()


def tokenize_text(text, stopwords, stemming=False):

    '''
    :param text: input text
    :param stopwords: list of stopwords to be filtered from text
    :param stemming: booklean, using stemming or not
    :return: list of tokens from the input text - token representation of the input text
    '''

    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)]

    spec_symbs = ['+', ':', '|', '/', '.', ',']
    tokens_ = []

    for token in tokens:

        token = ''.join(symb for symb in token if symb not in spec_symbs)

        if re.search('[%0-9a-zA-Za-яA-Я]', token) and token.find('.') == -1 and re.search('-', token) is None:
            tokens_.append(token.lower())

        elif re.search('-', token):
            for tok in token.split('-'):
                tokens_.append(tok.lower())

    if stemming:
        stemmed = [global_stemmer.stem(t) for t in tokens_]
    else:
        stemmed = tokens_

    tokens_out = []
    for token in stemmed:
        if token not in stopwords:
            tokens_out.append(token)

    return tokens_out
```

After this step each text was represented by a list of word tokens.

The next step was the replacement of each token in a list with a vector from one of the pretrained language models — Glove or fasttext.

The result of this operation was that each text was now represented by a list of semantically rich word vectors. We have put a restriction on the maximum length of the list — 50 words, the headline was concatenated with the beginning of the article's body.

In order to equalise the lenght of all vectors the padding operation has been performed. We've now got the feature tensor of our text corpus, each row represents a sequence of pretrained word vectors of the chosen dimension.

In [ ]:
```
def vectorize_text_NN(df_train, maxlen=50, lang_model='Glove', text='full', mo
de='fast'):

    '''
    function vectorizing texts according to NN logic
    :param df_train:
    :return: np array X - matrix of features for thew given text
    '''

    # using headlines and short_description as input X
    if text == 'full':
        df_train['text_full'] = df_train.title + " " + df_train.text

        if mode == 'fast':
            df_train['full_50'] = pd.Series(index=df_train.index)
            for i, row in df_train.iterrows():
                curr_word_lst = df_train['text_full'][i].split(' ')
                if len(curr_word_lst) > maxlen:
                    df_train['full_50'][i] = ' '.join(curr_word_lst[:maxlen])
                else:
                    df_train['full_50'][i] = df_train['text_full'][i]

    df = df_train.reset_index(drop=True)

    # vectorizing text
    if lang_model == 'Glove':
        stopws = stopwords.words('english')
    elif lang_model == 'ru':
        stopws = stopwords.words('russian')

    if lang_model == 'fasttext':
        lang_model_semantic = open_fasttext_model(path_to_model='models/wiki-n
ews-300d-1M-subword.vec')
    elif lang_model == 'Glove':
        lang_model_semantic = open_glove_model(path_to_model='tgnews/models/gl
ove.6B.100d.txt')
    elif lang_model == 'ru':
        lang_model_semantic = open_fasttext_model(path_to_model='tgnews/model
s/cc.ru.300.vec')

    lang_model_sem_dim = len(lang_model_semantic['.'])

    X = []

    for index, row in df.iterrows():

        if text == 'full':
            if mode == 'fast':
                curr_descr = df['full_50'][index].lower()
            else:
                curr_descr = df['text_full'][index].lower()
        elif text == 'title':
            curr_descr = df['title'][index].lower()
        elif text == 'text':
            curr_descr = df['text'][index].lower()
```

```
        curr_word_tokens = word_tokenize(curr_descr)
        curr_word_tokens = words_filtering(curr_word_tokens, stopws)

        entity_vector_sem = []
        for w in curr_word_tokens:

            if w in lang_model_semantic:
                entity_vector_sem.append(lang_model_semantic[w])
            else:
                entity_vector_sem.append(np.zeros(lang_model_sem_dim))

        X.append(entity_vector_sem)

    # # saving resulting np.array to pickle
    # with open('output/X_Glove.pkl', 'wb') as f:
    #     pickle.dump(X, f)

    X = list(sequence.pad_sequences(X, maxlen=maxlen))
    X = np.array(X)
    print('X.shape', X.shape)

    return X
```

Deep Neural Network Architecture

Since we had quite enough data for a neural network training we have decided to use a deep neural network (DNN) classifier in order to tell articles from and to define news categories.

Taking into account it was a algorithm contest an obvious choice maximising the solution's accuracy would be a SOTA NLP model architecture, namely some kind of large Transformer like BERT, but as we have mentioned before, these kinds of models would be too large and too slow to pass the restrictions imposed on hardware and on text processing speed. The other drawback is that such model's training would take a few days leaving me no time for model's fine-tuning.

We had to come up with a simpler architecture so we have implemented a lightweight neural net with an RNN (LSTM) layer taking the sequence of words embeddings representing texts and an Attention layer on top of it. The output of the network should be the class probabilities (binary classification for news filtering step and multiclass for news categories detection), so the upper part of the network was comprised by a set of fully connected layers.

Article Category Detection — Multiclass Classification

We shall be explaining the selection of particular DNN architecture using the multiclass classifier (detecting news categories) as an example because its objective is more challenging and its performance is independent of the threshold value used in the binary classifier to draw the margin between positive and negative classes.

The upper part of our neural net was made up of three Dense layers, the output layer had 7 units corresponding to the number of classes with the softmax activation function and categorical crossentropy as a loss function.

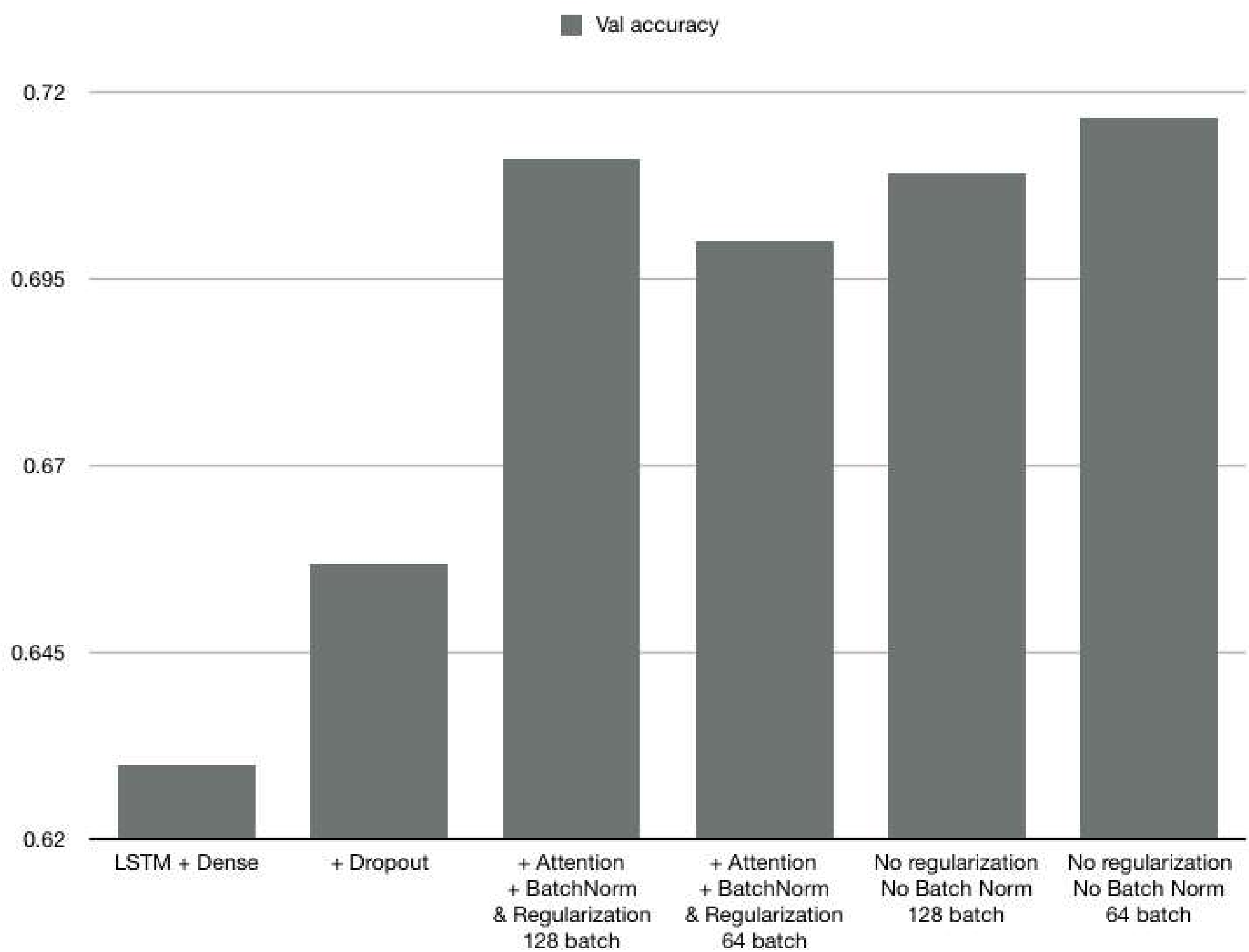
To train it we used the Dataset and applied a mapping logic in order to fit the initial articles categories.

In [ ]:
```
class LSTM_Attention_classifier_model:

    def __init__(self, maxlen, lang_model_sem_dim, n_classes):

        input_layer = keras.layers.Input(shape=(maxlen, lang_model_sem_dim))
        lstm_layer = keras.layers.LSTM(300, dropout=0.25, recurrent_dropout=0.
25, return_sequences=True)
        hidden = lstm_layer(input_layer)
        hidden = keras.layers.Dropout(0.25)(hidden)
        hidden = Attention(maxlen)(hidden)
        hidden = keras.layers.Dense(256, activation='relu')(hidden)  # kernel_
regularizer=regularizers.l2(0.01)
        hidden = keras.layers.Dropout(0.25)(hidden)
        hidden = keras.layers.Dense(128, activation='relu')(hidden)

        if n_classes > 2:
            self.classifier_type = 'MULTICLASS'
            output_layer = keras.layers.Dense(n_classes, activation='softmax')
(hidden)
            loss_func = 'categorical_crossentropy'
        elif n_classes == 2:
            self.classifier_type = 'BINARY'
            output_layer = keras.layers.Dense(1, activation='sigmoid')(hidden)
            loss_func = 'binary_crossentropy'
        else:
            print('Cannot compile a classifier for less than 2 classes, n_clas
ses:', n_classes)

        self.model = keras.models.Model(inputs=input_layer, outputs=output_lay
er)
        self.model.compile(loss=loss_func, optimizer='adam', metrics=['acc'])
        self.model.summary()


    def fit(self, x_train, x_val, y_train, y_val):

        training_history = self.model.fit(x_train, y_train, batch_size=32, epo
chs=30, validation_data=(x_val, y_val),
                                          callbacks=[keras.callbacks.EarlyStop
ping(monitor='val_loss', patience=5)])

        # printing training output and learning curves
        acc = training_history.history['acc']
        val_acc = training_history.history['val_acc']
        loss = training_history.history['loss']
        val_loss = training_history.history['val_loss']
        epochs = range(1, len(acc) + 1)

        plt.title('Training and validation accuracy')
        plt.plot(epochs, acc, 'red', label='Training acc')
        plt.plot(epochs, val_acc, 'blue', label='Validation acc')
        plt.legend()
        plt.figure()
        plt.title('Training and validation loss')
        plt.plot(epochs, loss, 'red', label='Training loss')
        plt.plot(epochs, val_loss, 'blue', label='Validation loss')
```

```
        plt.legend()
        plt.show()

        # saving model
        try:
            self.model.save(filepath=path_to_drive+'AttentionLSTM_model_{}.hdf
5'.format(self.classifier_type), overwrite=True, include_optimizer=True)
        except BaseException as e:
                print(e)


    def predict(self, X):

        '''Prediciting categories for telegram news with trained multiclass cl
assifier (DNN)'''

        y_probas = self.model.predict(X)

        return y_probas
```

WE would like to share the logic behind the NN model's architecture selection and hyperparameters fine-tuning.

1. The Dropout layers increase the generalisation ability of our model and prevent it from overfitting (Dropout layer randomly zeroes out the given percentage of weights at each update of the training phase). Model's overfitting without the Dropout layers is clearly demonstrated by the learning curves — the accuracy on the training set grows up to 95% while the accuracy on the validation dataset barely grows during training.
2. A BatchNormalisation layer applied on top of the LSTM-Attention construction normalized the activations after the dropout at each batch keeping the activation mean close to 0 and the activation standard deviation close to 1. It tends to increase test accuracy on the larger batch sizes and to decrease it on the smaller ones.
3. Regularization applied to Dense layers penalizes the extreme values of layer parameters.
4. Batch size selection can also affect a model's performance. The larger the size is the faster your model trains and the more precisely the gradient vector is calculated on each step. This results in noise reduction which makes the model more prone to converging to a local minimum so the batch size selection is usually a tradeoff between speed, memory consumption and model's performance. Values between 32 and 256 are the common choice, in our case the model showed top accuracy with batch size 64. Increasing batch size up to 512 or 1024 significantly decreases model's accuracy (by 2% and 4% accordingly)



Attention layer explained

It is worth saying a few words about the attention mechanism used in our model. The theory behind the applied approach is described in the arXiv paper by Raffel et al. This is a simplified model of attention for feed-forward neural networks addressing the RNN information flow problem for long sequences. Particularly, the attention layer provides an optimal transition to the fully connected layer, creating a context vector (an embedding for the sequence of input word vectors) as the weighted average of the hidden states of the input sequence with the weights representing the importance of the elements of the sequence. The explicit notation looks the following way:

$$c = \sum_{t=1}^{T} \alpha_t \, (h_t)$$

$$\alpha_i = \frac{\exp\,(e_t)}{\sum_{k=1}^{T} \exp\,(e_k)}$$

$$e_t = a(h_t)$$

where T is the length of sequence and a is a learnable function, namely a single hidden layer feed-forward network with the tanh activation function, jointly trained with the global model.

In [ ]:
```
class Attention(Layer):
    def __init__(self, step_dim,
                 W_regularizer=None, b_regularizer=None,
                 W_constraint=None, b_constraint=None,
                 bias=True, **kwargs):
        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')
        self.W_regularizer = regularizers.get(W_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)
        self.W_constraint = constraints.get(W_constraint)
        self.b_constraint = constraints.get(b_constraint)
        self.bias = bias
        self.step_dim = step_dim
        self.features_dim = 0
        super(Attention, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3
        self.W = self.add_weight(shape=(input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)

        self.features_dim = input_shape[-1]
        if self.bias:
            self.b = self.add_weight(shape=(input_shape[1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)
        else:
            self.b = None
        self.built = True

    def compute_mask(self, input, input_mask=None):
        return None

    def call(self, x, mask=None):
        features_dim = self.features_dim
        step_dim = self.step_dim
        eij = K.reshape(K.dot(K.reshape(x, (-1, features_dim)), K.reshape(self
.W, (features_dim, 1))), (-1, step_dim))
        if self.bias:
            eij += self.b
        eij = K.tanh(eij)
        a = K.exp(eij)
        if mask is not None:
            a *= K.cast(mask, K.floatx())
        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())
        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0], self.features_dim
```

```
    def get_config(self):
        config = {
            'step_dim': self.step_dim,
            'W_regularizer': self.W_regularizer,
            'b_regularizer': self.b_regularizer,
            'W_constraint': self.W_constraint,
            'b_constraint': self.b_constraint,
            'bias': self.bias
        }
        base_config = super(Attention, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

The drawback of this model is that it does not take the order within an input sequence into account but for our task of news headlines vectorization this is not as significant as for some sequence-to-sequence problems such as phrase translation.

There is a number of publications describing the more complicated attention mechanisms designed for seqence-to-sequence problems, a good start would be the Neural Machine Translation with Attention tutorial by Google Research, we would also recommend to check the Attention? Attention! post by Lilian Weng with an overview of attention mechanisms and their evolution and make sure you have read the original Bahdanau et al, 2015 paper. Among the more up-to-date LSTM + Attention papers of the post-Transformer era we would recommend reading recent Single-headed attention RNN by Stephen Merity, proving that the huge Transformers are not the only possible approach.

Text vectorization

In order to group articles first we should introduce some kind on metrics on the dataset. Since we used DNNs for texts processing and classification the most obvious way to get a text's vector would be taking the embedding obtained by passing the text through the pretrained multiclass DNN without the last layer. I used the 128 unit dense layer as the output to create texts embedding vectors.

As we found later this approach was not the optimal one — a significantly better performance in articles grouping was demonstrated when we switched to a simpler TF-IDF vectorization. This is quite explainable — while in articles category classification we needed to generalize the whole articles semantic meaning regardless of particular comma, full stop, how, what, etc keywors and even particular circumstances, the sequence of pretrained word vectors fed to the DNN was a suitable choice for the task. In articles grouping we are dealing with a different setting — like the technical keywords Android, C++, Python, Data Science, Data Structure, ML, so the classic TF-IDF (calculating the vector of n-gram frequencies in each text normalized by the n-gram frequencies in the whole corpus) is the approach loosing less valuable information. To fight the TF-IDF output matrix sparsity I have applied an SVD decomposition compressing each text's vector to the selected dimension.

In [ ]:
```
def vectorize_texts_tf_idf(data_set, ngram, using_stemming=True):
    '''
    :param data_set: list of texts
    :param ngram: ngram range used
    :param using_stemming: boolean
    :return: X - matrix of texts embeddings
    '''

    stopws = stopwords.words('english')

    corpus = []
    for text in data_set:
        tokens = tokenize_text(text, stopws, using_stemming)
        text_string = ''
        for word in tokens:
                text_string += word + ' '
        corpus.append(text_string)

    vectorizer = TfidfVectorizer(input='content', encoding='utf-8', analyzer=
'word', max_df=0.4, max_features=20000, min_df=0.00001, use_idf=True, ngram_ra
nge=ngram)
    t0 = time.time()
    X0 = vectorizer.fit_transform(corpus)
    print("vectorization done in", (time.time() - t0))
    print("n_samples: %d, n_features: %d" % X0.shape)

    if X0.shape[1] > 50:
        svd_flag = 1
        if X0.shape[1] >= 1000:
            n_components_real = 1000
    else:
        svd_flag = 0

    if svd_flag == 1:
        t0 = time.time()
        normalizer = Normalizer(copy=False)
        svd = TruncatedSVD(n_components=n_components_real, algorithm='randomiz
ed', n_iter=5, random_state=None, tol=0.0)
        lsa = make_pipeline(svd, normalizer)
        X = lsa.fit_transform(X0)
        print("svd done in: ", time.time() - t0)
    else:
        X = X0

    return X
```

The Grouping Algorithm

Ok, now we’ve finally have got an index of all articles and can group them in threads by distance between different samples.

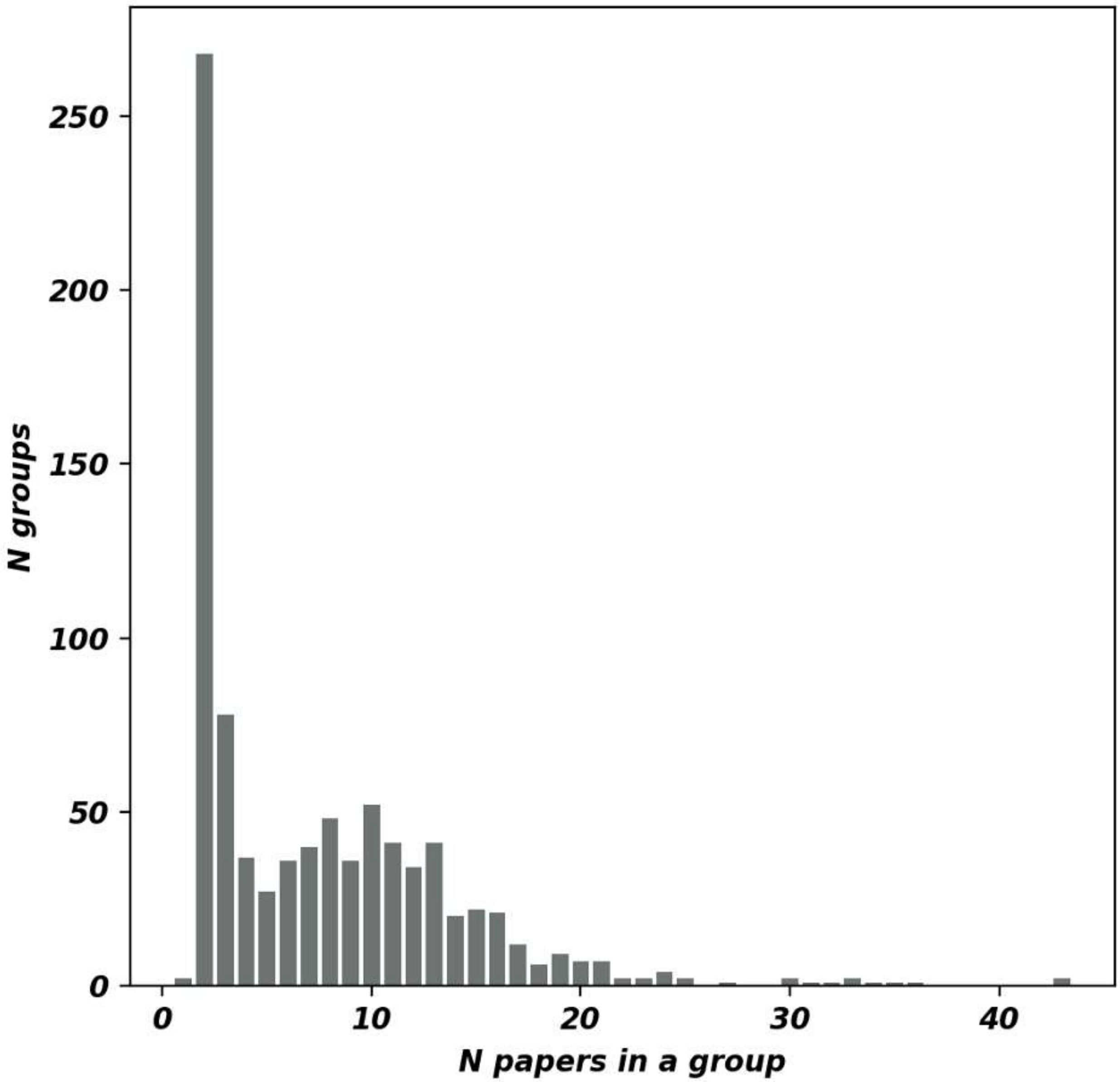
We did not have enough time to reflect on some more sophisticated approaches and just created a cycle over all papers in each articles category (we can take advantage of the pervious algorithm step to partition our dataset by artciles categories) checking for their neighbors in radius r_start, r_start was selected empirically in such a way that it took a little more papers than the actual thread contained. Then we have calculated an empirical functional variative_criterium_norm — the normalized Levenstein distance between the texts within the group — and decreased the search radius r_curr iteratively until this functional became less than the empirically found constraint or the search radius size hit r_min constraint. If there were no articles in the given area, we have increased the search radius until we found some neighbors and then switched to the radius decreasing branch. The idea behind this iterative search was that similar articles have partly similar headlines (some entities like subject, object and sometimes verbs are invariant over the articles thread). The other details of the developed algoritms are easier to see from the code snippet.

In [ ]:
```
def group_news(df, tree, embs, debug=False, r_min=0.01, r_max=4.0, r_start=0.2
5, r_step=0.1, vc_min=0.01, vc_max=0.04, delta_max=0.01):

    '''
    NEWS GROUPING LOGIC
    :param df: dataframe with news
    :param tree: KDtree built on embs for the data in df
    :param embs: embeddings for each text created with the NN used for multicl
ass classification
    :return:
    '''

    total = len(df)
    grouped_ind = []  # creating a list to store papers already used in other
 groups
    news_groups = []  # groups of news
    ungrouped_list = []  # herea will be papers ungrouped due to the strict lo
gic

    for index, rows in df.iterrows():

        if index in grouped_ind:
            pass

        else:
            if debug:
                print(index, '/', total)

            r_curr = r_start
            variative_criterium_norm = vc_min + 0.0000001
            flag_single = 0
            delta = 0

            # defining the search radius individually for each news group
            while vc_max > variative_criterium_norm > vc_min and r_max > r_cur
r > r_min and delta < delta_max:

                ind = tree.query_radius(np.array([embs[index]]), r=r_curr)

                # checking that we have more than 0 results for the current qu
ery
                if len(ind[0]) > 1.0:

                    # this guarantees that we shall not go to the second curcl
e of radius increase
                    if flag_single > 0.1:
                        flag_single = 10

                    variative_criterium = 0.

                    # clac variative criterium
                    for j in range(len(ind[0])):
                        if debug:
                            print(df.title[ind[0][j]])

                        if len(ind[0]) > j+1:
```

```
                            a = df.title[ind[0][j]]
                            b = df.title[ind[0][j + 1]]
                            delta = edlib.align(a, b)['editDistance'] / np.ave
rage([len(a), len(b)])
                            variative_criterium += delta

                    variative_criterium_norm = variative_criterium / len(ind[0
])
                    r_curr -= r_step
                    if debug:
                        print('VC:', variative_criterium_norm, '\n', 'r:', r_c
urr, '\n' * 2)

                else:
                    #print(df.title[ind[0][0]])
                    if flag_single < 5:
                        flag_single +=1
                        variative_criterium = vc_min + 0.1
                        r_curr += 5*r_step
                        if debug:
                            print('VC:', variative_criterium_norm, '\n', 'r:',
r_curr, '\n' * 2)
                    else:
                        break


            # SIMPLISTIC groups grouping logic - ver 3.0
            if len(ind[0]) > 0.1:
                news_groups.append(list(ind[0]))
                grouped_ind.extend(list(ind[0]))

    # Preparing output
    groups_out = []
    groups_titles = []

    for group in news_groups:

        groups_out.append({'title': df['title'][group[0]], 'articles': [df['fi
d'][paper] for paper in group]})
        titles = [df.title[paper] for paper in group]
        groups_titles.append(titles)

    return news_groups, groups_out, groups_titles
```

There are 7 hyperparameters controlling the algorithm: r_min, r_max — these control the size of the query area, r_start, r_step — controlling the query area dynamics and vc_min, vc_max, delta_max, controlling the value of normalized Levenshtein distance within a group—this defines the variance of news headlines in a group. These hyperparameters should be tuned after you have chosen the vectorization parameters like the final text embedding size (n_components in SVD) and the range of n_grams used (I used 1-3).

It is not really obvious how we can estimate a “proper” articles grouping, so we did not spend too much time playing with hyperparameters after we got a reasonable grouping result — our intention was to suggest a working approach to solve the problem with the given constraints. Clearly there is some space for improvement like checking if we can merge some of the groups and filtering out some occasional noise. Actually in order to get a reasonable number and density of groups the grouping hyperparameters should be fine tuned to each dataset, so there could be an outer cycle implementing a kind of Randomized search on them. One of the ways to get an idea of the particular grouping we got and to estimate its quality is to check the groups size distribution histogram.


250
200
150
100
50
0
N groups
0
10
20
30
40
N papers in a group

In fact, the described approach could be regarded as a relative of the DBSCAN clustering. Execution time varies depending on the hyperparameters chosen for the dataset and the structure of data, the typical values are from 8.5 sec / 1000 papers to 25 sec / 1000 papers including the vectorization time defined by the expensive SVD operation.

Here are the dummy results for the ANDROID category articles.

Output

1. Why Should You Learn Android App Development?
2. How to Use Canvas API in Android Apps?
3. How to Create a Credit Card Form in Android?
4. Overview of WorkManager in Android Architecture Components
5. Strikethrough Text in Android
6. Understanding Density Independence Pixel: sp, dp, dip in Android
7. Snackbar in Android using Jetpack Compose
8. How to Detect Text Type Automatically in Android?
9. How to Make Substring of a TextView Clickable in Android?
10. How to Create New ImageView Dynamically on Button Click in Android?
11. What is “Don’t Keep Activities” in Android?
12. How to Build a Number Shapes Android App in Android Studio?
13. How to Create WhatsApp Stories View in Android?
14. How to Add Florent LongShadow to Android App?
15. What is NDK in Android?
16. How to Add SlantedTextView in Android App?
17. How to Create Blink Effect on TextView in Android?
18. Typing Animation Effect in Android'
19. Explode Animation in Android
20. Zoom Scroll View in Android
21. How to Create Balloon Toast Message in Android?
22. How to Create Star Animation in Android?
23. GravityView in Android
24. How to Add Vector Assets in Android Studio?
25. ConstraintLayout in Android
26. Endless RecyclerView in Android
27. How to Enable Full-Screen Mode in Android?
28. Implementation of HtmlTextView in Android
29. How to Use SnapHelper in RecyclerView in Android?
30. How to Create Circular Determinate ProgressBar in Android?
31. Tinder Swipe View with Example in Android
32. How to Add Fade and Shrink Animation in RecyclerView in Android?
33. How to Fix “Android studio logcat nothing to show” in Android Studio?
34. Implement Splash Screen and Authentication in Social Media Android App
35. Implementing Edit Profile Data Functionality in Social Media Android App
36. How to Retrieve Blog On Home Page in Social Media Android App?
37. How to Add Blogs in Social Media Android App?
38. How to Add Share Button in Toolbar in Android?
39. How to Use Flows in Android ConstraintLayout to Build Complex Layouts?
40. Guidelines in Android ConstraintLayout