Text Mining and Deep Learning
Suggestion Mining from Online Reviews and Forums

Duy Nguyen

January 24, 2019

Contents

1 Introduction

Suggestion mining can be defined as the extraction of suggestions from unstructured text, where the term 'suggestions' refers to the expressions of tips, advice, recommendations etc. Consumer opinions towards commercial entities like brands, services, and products are generally expressed through online reviews, blogs, discussion forums, or social media platforms. These opinions largely express positive and negative sentiments towards a given entity, but also tend to contain suggestions for improvising the entity or tips to the fellow consumers. Traditional opinion mining systems mainly focus on automatically calculating the sentiment distribution towards an entity of interest by means of Sentiment Analysis methods. A suggestion mining component can extend the capabilities of traditional opinion mining systems, which can then cater to additional applications. Such systems can empower both public and private sectors by extracting the suggestions which are spontaneously expressed on various online platforms, enabling the organisations to collect suggestions from much larger and varied sources of opinions than the traditional suggestion box or online feedback forms.

2 Dataset Analysis

	Subtask-A	Subtask-B	Percent	Vocabulary
Suggestions	1986	1986	24.6	NA
Non-suggestions	6067	6067	75.4	NA
Total	8053	8053	100	9072

Subtask A To perform domain specific suggestion mining, i.e., suggestion forum for windows platform developers.

SubTask B To perform cross domain suggestion mining, i.e.,model trained on suggestion forum dataset will be evaluated on hotel review.

3 Problem Analysis

3.1 Sample Paragraphs

These are the sample paragraphs which I use to analyse the level to cover in deduction a comments is a suggestion or not.

In updating the unit's firmware: When I first purchased the unit I downloaded the firmware update that was available on their website. The upgrade went smoothly and was easy to do if you can follow directions. I suggest that doing the upgrade to be sure you have the best chance at trouble free operation.

Below is the levels of context, sentence, phrase and word tokens. Which can be used to propose suitable method for suggestion prediction.

This is also has the same structure to analyse: The storage capacity is great. I don't think I will ever use the full 30GB. I currently have 1100 songs encoded

mostly at 196 kpbs and I haven't even used 10GB of storage yet. I'm starting to listen to CD's that I haven't listened to in years now that I have them all handy in one place.

During many issues prone by Python, Anaconda and Windows system. I choose CNN as the main method for suggestion prediction.

4 Choice of Method

4.1 Fasttext

Training supervised model using fasttext got problem with Windows system, which need more time for sorting out the issue. After investigation I found that this issue is not occurred in Linux environment.

4.2 CNN model

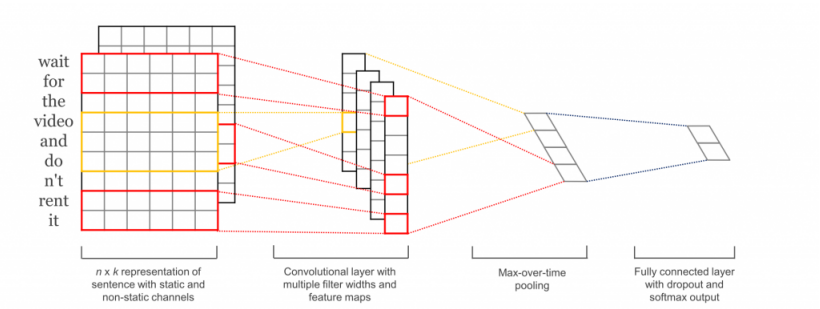

Figure 1: Convolutional Neural Networks

The first layers embeds words into low-dimensional vectors. The next layer performs convolutions over the embedded word vectors using multiple filter sizes. For example, sliding over 3, 4 or 5 words at a time. Next, we max-pool the result of the convolutional layer into a long feature vector, add dropout regularization, and classify the result using a softmax layer.

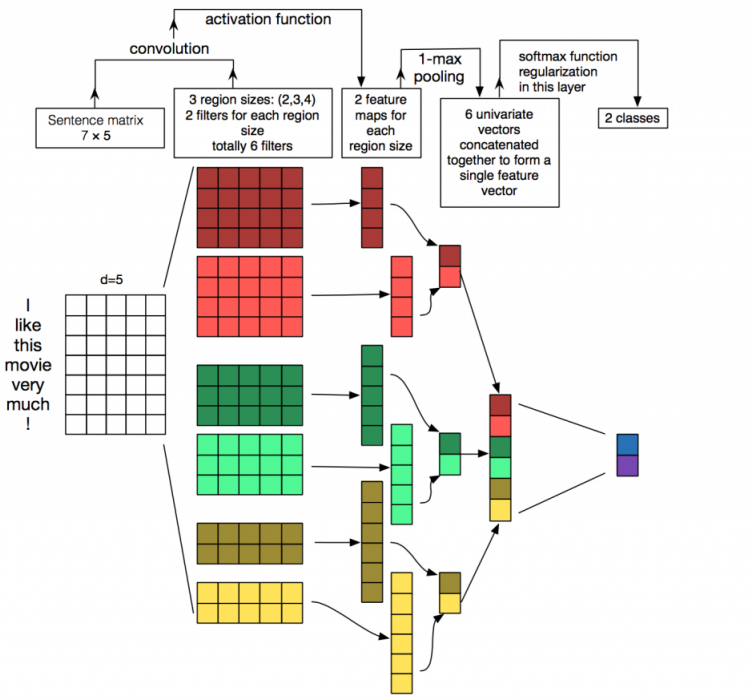

Figure 2: Convolutional Neural Networks for sentences

I will not used pre-trained word2vec vectors for my word embeddings. Instead, we learn embeddings from scratch. Because I have an issue with importing word2vec on my computer. After 2 times of importing for more than 4 hours each. My computer crash and I have no other choices.

5 Implementation

5.1 Data and Preprocessing

Due to unbalance dataset, the non-suggestion sentences have three times more than suggestion sentences. So I have to use method of under-sampling and over-sampling.

5.1.1 Under-sampling

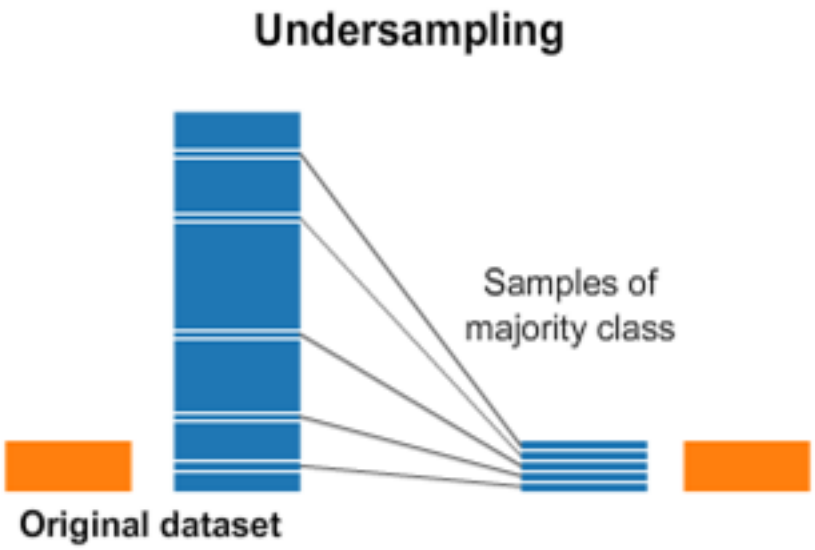

Figure 3: Under-Sampling

Solution for this is just shuffle the majority class and take 1/3 of the whole set of sentences. Or I just remove the redundant part of the majority class.

```
def undersampling(minor_class_file, major_class_file):
    """
    Undersampling dataframe object
    """
    minor_class_file = load_data(minor_class_file)
    major_class_file = load_data(major_class_file)
    shuffle(major_class_file)
    # Tokenize the sorted list
    tokenized_minor = [word_tokenize(i) for i in major_class_file]

    removing_sentences = len(major_class_file) - len(minor_class_file)
    major_class_file = major_class_file[0:len(minor_class_file)]
    return major_class_file
```

5.1.2 Over-sampling

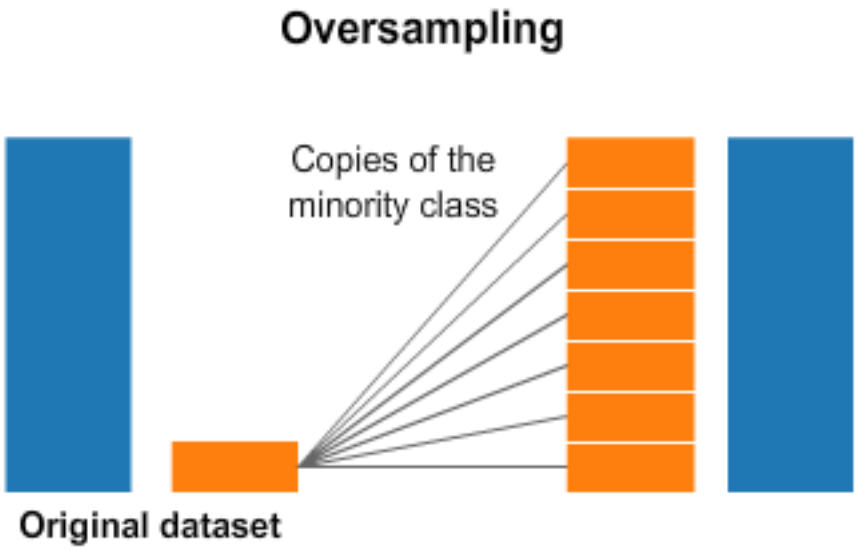

Figure 4: Over-Sampling

Actually NLP is one of the most common areas in which resampling of data is needed as there are many text classification tasks dealing with imbalanced problem (think of spam filtering, insulting comment detection, article classification, etc.). But SMOTE seem to be problematic here for some reasons:

- SMOTE works in feature space. It means that the output of SMOTE is not a synthetic data which is a real representative of a text inside its feature space.
- On one side SMOTE works with KNN and on the other hand, feature spaces for NLP problem are dramatically huge. KNN will easily fail in those huge dimensions.

I do another oversampling which is on the text (so more intuitive) and is kind of SMOTE.

1. Ignore the major class. Get a length distribution of all sentences in minor class so that we generate new samples according the the true sentence length (number of words/phrases). We assume we want to make the size of class triple (so producing k=2 synthetic sentences per original document).
2. Generate a sequence of n random integers according to that distribution. It's used for determining the length of new synthetic sentence.

3. For each sentence: Choose one integer from random length sequence and m random document whose length is close to the integer. Put tokens of all m sentences in a set and randomly choose n tokens k times. these are your k new sentences.

```
def oversampling(minor_class_file, major_class_file, div, mod):
    """
    Oversampling dataframe object
    """
    minor_class_file = load_data(minor_class_file)
    major_class_file = load_data(major_class_file)

    # Sort list according length of elements
    minor_class_file = sorted(minor_class_file, key=len)

    # Tokenize the sorted list
    tokenized_minor = [word_tokenize(i) for i in minor_class_file]
    length_tk = len(tokenized_minor)

    # Build vocabulary
    max_sentence_length = max([len(x.split(" ")) for x in minor_class_file])
    adding_sentences = len(major_class_file) - len(minor_class_file)

    num_of_neighbor = 5
    for i in range(adding_sentences):
        rand_num = random.randint(1, max_sentence_length)

        # Find position of tokenized_minor
        pos = int((length_tk*rand_num)/max_sentence_length)

        # Find neighbor os rand_num
        neighbors=[]
        neighbors = neighbor_of_rand(pos, length_tk, num_of_neighbor)

        # Create a list of tokens
        list_tk = []
        for k in neighbors:
            list_tk = list_tk + tokenized_minor[k-1]

        test = len(tokenized_minor[neighbors[0]])
        shuffle(list_tk)

        # added_sen = random.sample(list_tk, len(tokenized_minor(pos)))
        list_tk = list_tk[:test]
        added_sentence = ' '.join(list_tk)
```

```
            minor_class_file.append(added_sentence)
        shuffle(minor_class_file)
        return minor_class_file
```

5.2 Main CNN implementation

```
import tensorflow as tf
import numpy as np

class TextCNN(object):
    """
    A CNN for text classification.
    Uses an embedding layer, followed by a
    convolutional, max-pooling and softmax layer.
    """
    def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters):
```

List of arguments:

1. sequence length – The length of our sentences. Remember that we padded all our sentences to have the same length.
2. num classes – Number of classes in the output layer, two in our case (suggestion and non-suggestion).
3. num classes – Number of classes in the output layer, two in our case (suggestion and non-suggestion).
4. vocab size – The size of our vocabulary. This is needed to define the size of our embedding layer, which will have shape [vocabulary size, embedding size].
5. embedding size – The dimensionality of our embeddings.
6. filter sizes – The number of words we want our convolutional filters to cover. We will have num filters for each size specified here. For example, [3, 4, 5] means that we will have filters that slide over 3, 4 and 5 words respectively, for a total of 3 * num filters filters.
7. num filters – The number of filters per filter size.

5.3 Embedding Layer

The first layer we define is the embedding layer, which maps vocabulary word indices into low-dimensional vector representations. It's essentially a lookup table that we learn from data.

```
with tf.device('/cpu:0'), tf.name_scope("embedding"):
    W = tf.Variable(
        tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0),
        name="W")
    self.embedded_chars = tf.nn.embedding_lookup(W, self.input_x)
    self.embedded_chars_expanded =
     tf.expand_dims(self.embedded_chars, -1)
```

tf.name scope creates a new Name Scope with the name "embedding". The scope adds all operations into a top-level node called "embedding" so that you get a nice hierarchy when visualizing your network in TensorBoard.

W is our embedding matrix that we learn during training. We initialize it using a random uniform distribution. tf.nn.embedding lookup creates the actual embedding operation. The result of the embedding operation is a 3-dimensional tensor of shape [None, sequence length, embedding size].

TensorFlow's convolutional conv2d operation expects a 4-dimensional tensor with dimensions corresponding to batch, width, height and channel. The result of our embedding doesn't contain the channel dimension, so we add it manually, leaving us with a layer of shape [None, sequence length, embedding size, 1].

5.4 Convolution and Max-Pooling Layers

Now we're ready to build our convolutional layers followed by max-pooling. Remember that we use filters of different sizes. Because each convolution produces tensors of different shapes we need to iterate through them, create a layer for each of them, and then merge the results into one big feature vector.

```
pooled_outputs = []
for i, filter_size in enumerate(filter_sizes):
    with tf.name_scope("conv-maxpool-%s" % filter_size):
        # Convolution Layer
        filter_shape = [filter_size, embedding_size, 1, num_filters]
        W = tf.Variable(tf.truncated_normal(filter_shape,
        stddev=0.1), name="W")
        b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
        conv = tf.nn.conv2d(
            self.embedded_chars_expanded,
            W,
            strides=[1, 1, 1, 1],
            padding="VALID",
            name="conv")
        # Apply nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
        # Max-pooling over the outputs
        pooled = tf.nn.max_pool(
            h,
```

```
            ksize=[1, sequence_length - filter_size + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name="pool")
        pooled_outputs.append(pooled)

# Combine all the pooled features
num_filters_total = num_filters * len(filter_sizes)
self.h_pool = tf.concat(3, pooled_outputs)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])
```

Here, W is our filter matrix and h is the result of applying the nonlinearity to the convolution output. Each filter slides over the whole embedding, but varies in how many words it covers. "VALID" padding means that we slide the filter over our sentence without padding the edges, performing a narrow convolution that gives us an output of shape [1, sequence length - filter size + 1, 1, 1]. Performing max-pooling over the output of a specific filter size leaves us with a tensor of shape [batch size, 1, 1, num filters]. This is essentially a feature vector, where the last dimension corresponds to our features. Once we have all the pooled output tensors from each filter size we combine them into one long feature vector of shape [batch size, num filters total]. Using -1 in tf.reshape tells TensorFlow to flatten the dimension when possible.

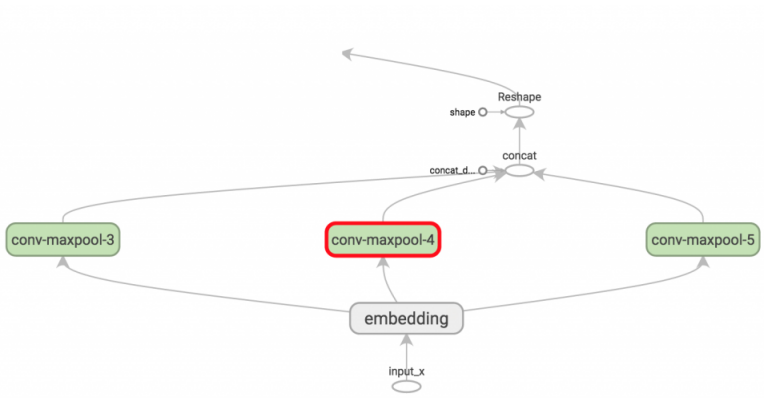

Figure 5: Convolutional diagram

5.5 Dropout Layer

Dropout is the perhaps most popular method to regularize convolutional neural networks. The idea behind dropout is simple. A dropout layer stochastically "disables" a fraction of its neurons. This prevent neurons from co-adapting and forces them to learn individually useful features. The fraction of neurons we keep enabled is defined by the dropout keep prob input to our network. We

set this to something like 0.5 during training, and to 1 (disable dropout) during evaluation.

```
# Add dropout
with tf.name_scope("dropout"):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)
```

5.6 Scores and Predictions

Using the feature vector from max-pooling (with dropout applied) we can generate predictions by doing a matrix multiplication and picking the class with the highest score. We could also apply a softmax function to convert raw scores into normalized probabilities, but that wouldn't change our final predictions.

```
with tf.name_scope("output"):
    W = tf.Variable(tf.truncated_normal
      ([num_filters_total, num_classes], stddev=0.1), name="W")
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
    self.predictions = tf.argmax(self.scores, 1, name="predictions")
```

Here, tf.nn.xwplusb is a convenience wrapper to perform the Wx + b matrix multiplication.

5.7 Precision Recall, F1 score, Loss and Accuracy

Using our scores we can define the loss function. The loss is a measurement of the error our network makes, and our goal is to minimize it. The standard loss function for categorization problems it the cross-entropy loss.

```
# Calculate mean cross-entropy loss
with tf.name_scope("loss"):
    losses = tf.nn.softmax_cross_entropy_with_logits(self.scores,
     self.input_y)
    self.loss = tf.reduce_mean(losses)
```

Here, tf.nn.softmax cross entropy with logits is a convenience function that calculates the cross-entropy loss for each class, given our scores and the correct input labels. We then take the mean of the losses. We could also use the sum, but that makes it harder to compare the loss across different batch sizes and train/dev data.

We also define an expression for the accuracy, which is a useful quantity to keep track of during training and testing.

```
# Calculate Accuracy
with tf.name_scope("accuracy"):
    correct_predictions = tf.equal(self.predictions,
     tf.argmax(self.input_y, 1))
    self.accuracy = tf.reduce_mean(tf.cast
     (correct_predictions, "float"), name="accuracy")
```

I can also add F1 score and Precision recall metrics, which help to evaluate the trained model.

```
# Print accuracy if y_test is defined
if y_test is not None:
    # print("all_predictions: ", all_predictions)
    # print("y_test: ", y_test)
    tn, fp, fn, tp = confusion_matrix(y_test, all_predictions).ravel()
    print("True Negative :", tn)
    print("False Positive:", fp)
    print("False Negative:", fn)
    print("True Positive :", tp)
    print("Precision      : %8.2f" %
    (sk.metrics.precision_score(y_test, all_predictions)))
    print("Recall         : %8.2f" %
    (sk.metrics.recall_score(y_test, all_predictions)))
    print("f1_score       : %8.2f" %
    (sk.metrics.f1_score(y_test, all_predictions)))
    correct_predictions = float(sum(all_predictions == y_test))
    print("Total number of test examples: {}".format(len(y_test)))
    print("Accuracy       : {:g}".format
    (correct_predictions/float(len(y_test))))
```

5.8 Visualizing the network

Using the feature vector from max-pooling (with dropout applied) we can generate predictions by doing a matrix multiplication and picking the class with the highest score. We could also apply a softmax function to convert raw scores into normalized probabilities, but that wouldn't change our final predictions.

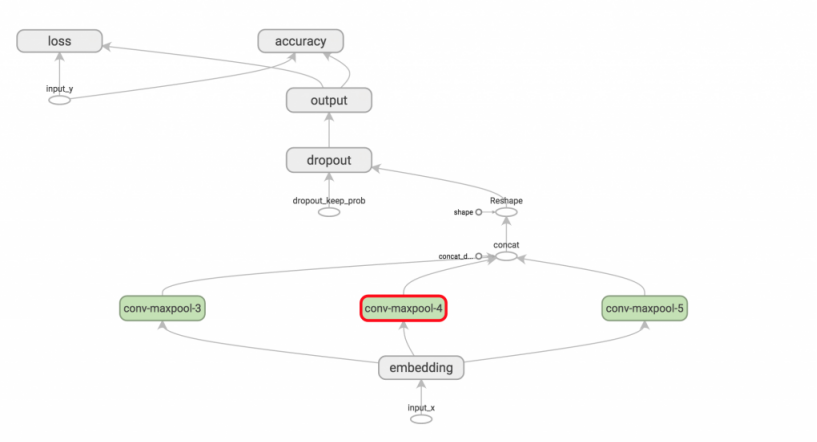

Figure 6: Visualized tensorboard network

5.9 Training Procedure

Using session and graph in tensorboard.

```
with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
```

The allow soft placement setting allows TensorFlow to fall back on a device with a certain operation implemented when the preferred device doesn't exist. For example, if our code places an operation on a GPU and we run the code on a machine without GPU, not using allow soft placement would result in an error. If log device placement is set, TensorFlow log on which devices (CPU or GPU) it places operations. That's useful for debugging. FLAGS are command-line arguments to our program.

5.10 Instantiating the CNN and minimizing the loss

When we instantiate our TextCNN models all the variables and operations defined will be placed into the default graph and session we've created above.

```
cnn = TextCNN(
    sequence_length=x_train.shape[1],
    num_classes=2,
    vocab_size=len(vocabulary),
```

```
    embedding_size=FLAGS.embedding_dim,
    filter_sizes=map(int, FLAGS.filter_sizes.split(",")),
    num_filters=FLAGS.num_filters)
```

Next, we define how to optimize our network's loss function. TensorFlow has several built-in optimizers. We're using the Adam optimizer.

```
global_step = tf.Variable(0, name="global_step", trainable=False)
optimizer = tf.train.AdamOptimizer(1e-4)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars,
global_step=global_step)
```

Here, train op here is a newly created operation that we can run to perform a gradient update on our parameters. Each execution of train op is a training step. TensorFlow automatically figures out which variables are "trainable" and calculates their gradients. By defining a global step variable and passing it to the optimizer we allow TensorFlow handle the counting of training steps for us. The global step will be automatically incremented by one every time you execute train op.

5.11 Summaries

TensorFlow has a concept of a summaries, which allow you to keep track of and visualize various quantities during training and evaluation. For example, you probably want to keep track of how your loss and accuracy evolve over time. You can also keep track of more complex quantities, such as histograms of layer activations. Summaries are serialized objects, and they are written to disk using a SummaryWriter.

```
# Output directory for models and summaries
timestamp = str(int(time.time()))
out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs", timestamp))
print("Writing to {}\n".format(out_dir))

# Summaries for loss and accuracy
loss_summary = tf.scalar_summary("loss", cnn.loss)
acc_summary = tf.scalar_summary("accuracy", cnn.accuracy)

# Train Summaries
train_summary_op = tf.merge_summary([loss_summary, acc_summary])
train_summary_dir = os.path.join(out_dir, "summaries", "train")
train_summary_writer = tf.train.SummaryWriter
(train_summary_dir, sess.graph_def)

# Dev summaries
```

```
dev_summary_op = tf.merge_summary([loss_summary, acc_summary])
dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
dev_summary_writer = tf.train.SummaryWriter
 (dev_summary_dir, sess.graph_def)
```

Here, we are separately keeping track of summaries for training and evaluation. In our case these are the same quantities, but you may have quantities that you want to track during training only (like parameter update values). tf.merge summary is a convenience function that merges multiple summary operations into a single operation that we can execute.

5.12 Checkpointing

Another TensorFlow feature you typically want to use is checkpointing – saving the parameters of your model to restore them later on. Checkpoints can be used to continue training at a later point, or to pick the best parameters setting using early stopping. Checkpoints are created using a Saver object.

```
# Checkpointing
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
# Tensorflow assumes this directory already exists
so we need to create it
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.all_variables())
```

5.13 Initializing the variables

Before we can train our model we also need to initialize the variables in our graph.

```
sess.run(tf.initialize_all_variables())
```

The initialize all variables function is a convenience function run all of the initializers we've defined for our variables. You can also call the initializer of your variables manually. That's useful if you want to initialize your embeddings with pre-trained values for example.

5.14 Defining a single training step

Let's now define a function for a single training step, evaluating the model on a batch of data and updating the model parameters.

```
def train_step(x_batch, y_batch):
    """
    A single training step
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
    }
    _, step, summaries, loss, accuracy = sess.run(
        [train_op, global_step, train_summary_op,
        cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format
    (time_str, step, loss, accuracy))
    train_summary_writer.add_summary(summaries, step)
```

feed dict contains the data for the placeholder nodes we pass to our network. You must feed values for all placeholder nodes, or TensorFlow will throw an error. Another way to work with input data is using queues, but that's beyond the scope of this post.

Next, we execute our train op using session.run, which returns values for all the operations we ask it to evaluate. Note that train op returns nothing, it just updates the parameters of our network. Finally, we print the loss and accuracy of the current training batch and save the summaries to disk. Note that the loss and accuracy for a training batch may vary significantly across batches if your batch size is small. And because we're using dropout your training metrics may start out being worse than your evaluation metrics.

We write a similar function to evaluate the loss and accuracy on an arbitrary data set, such as a validation set or the whole training set. Essentially this function does the same as the above, but without the training operation. It also disables dropout.

```
def dev_step(x_batch, y_batch, writer=None):
    """
    Evaluates model on a dev set
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: 1.0
    }
    step, summaries, loss, accuracy = sess.run(
        [global_step, dev_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
```

```
    time_str = datetime.datetime.now().isoformat()
    print("{}: step {}, loss {:g}, acc {:g}".format
    (time_str, step, loss, accuracy))
    if writer:
        writer.add_summary(summaries, step)
```

5.15 Training loop

Finally, we're ready to write our training loop. We iterate over batches of our data, call the train step function for each batch, and occasionally evaluate and checkpoint our model:

```
# Generate batches
batches = data_helpers.batch_iter(
    zip(x_train, y_train), FLAGS.batch_size, FLAGS.num_epochs)
# Training loop. For each batch...
for batch in batches:
    x_batch, y_batch = zip(*batch)
    train_step(x_batch, y_batch)
    current_step = tf.train.global_step(sess, global_step)
    if current_step % FLAGS.evaluate_every == 0:
        print("\nEvaluation:")
        dev_step(x_dev, y_dev, writer=dev_summary_writer)
        print("")
    if current_step % FLAGS.checkpoint_every == 0:
        path = saver.save(sess, checkpoint_prefix,
        global_step=current_step)
        print("Saved model checkpoint to {}\n".format(path))
```

Here, batch iter is a helper function I wrote to batch the data, and tf.train.global step is convenience function that returns the value of global step.

6 User Manual

6.1 Configuration Python and Anaconda

This step requires a lot of effort and time, which can frustrate the working process. But in general, I learnt instructions from the main anaconda and Python support website to configure.

In order to install environment for tensorflow 1.12, I need to use Python 3.6 instead of Python 3.7. Because the missing support from Python 3.7.

Create a virtual environment in anaconda command prompt. Specifying the version is optional. conda create -n [envname] python=[pythonversion]

Activate the virtual environment source activate [envname]

Install all your packages conda install [packagename] pip install [packagename]

6.2 Preprocessing

Script to run: SuggestionClassifier.py Take the latest version of the suggestion dataset (V1.4Training.csv) and put it in the same directory with SuggestionClassifier.py file.

For undersampling, please change the parameters for undersampling as below.

```
# Misc Parameters
tf.flags.DEFINE_boolean("over_sampling", False, "Allow oversampling minor class"
tf.flags.DEFINE_boolean("under_sampling", True, "Allow undersampling major class
```

For oversampling, please change the parameters for undersampling as below.

```
# Misc Parameters
tf.flags.DEFINE_boolean("over_sampling", True, "Allow oversampling minor class")
tf.flags.DEFINE_boolean("under_sampling", False, "Allow undersampling major clas
```

Run this python script. It will read csv file, create two equal dataset of different classes in two files (suggestion.txt and non suggestion.txt).

The class which is used for undersampling or oversampling will be preprocessed. And the other class will be preprocessed in the next step with the same method. These method includes lowercase, removing meaningless special characters, long space.

For removing the whole blank sentence or stop words, I manually use online tool to process it.

6.3 Training

Script to run: train.py This script will use the two dataset file to train our model. Output will appear in "runs" folder.

6.4 Evaluation model

Script to run: eval.py Before running this script, we need to configure parameter for it. Options for these parameter are:

```
--eval_train --checkpoint_dir="./runs/1547031503/checkpoints/"
```

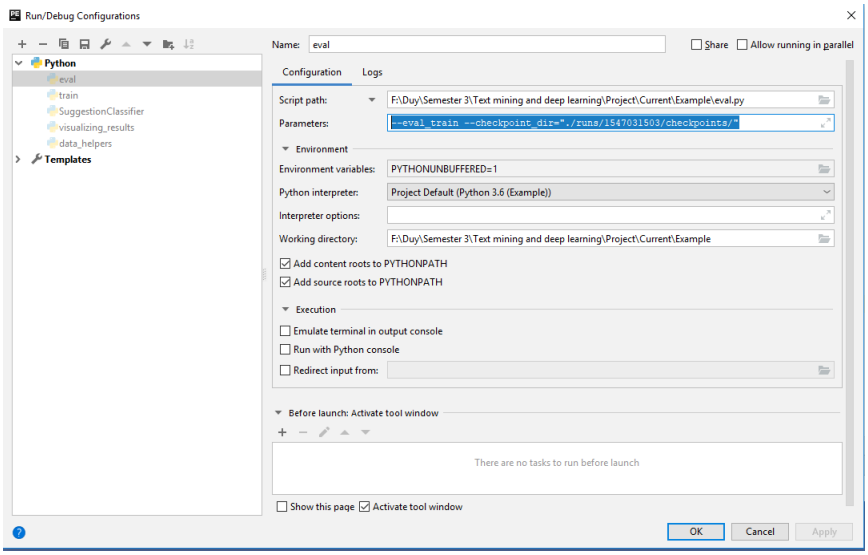

Figure 7: Evaluation model parameters

6.5 Visualizing Results in TensorBoard

Our training script writes summaries to an output directory, and by pointing TensorBoard to that directory we can visualize the graph and the summaries we created.

```
conda env list
activate env_name
tensorboard --logdir "PATH_TO_CODE/runs/1547021216/summaries/"

# Access this local web site to run tensorboard
http://localhost:6006/#scalars
```

Running the training procedure with default parameters (128-dimensional embeddings, filter sizes of 3, 4 and 5, dropout of 0.5 and 128 filters per filter size) results in the following loss and accuracy plots (blue is training data, red is 10

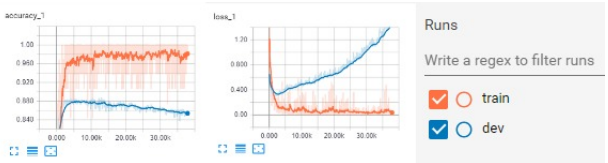

Figure 8: Accuracy - Loss

1. The labels consists of sequential numbers.Training metrics are not smooth because I use small batch sizes (red line).
2. Because dev accuracy is below training accuracy it seems like our network is overfitting the training data.

7 Results

Run	PreP	Sen	Acc	Precision	Recall	F1Score	BatchS	Time	EmptySen	Stopword
1503	Over	13444	0.966	0.96	0.98	0.97	32	10	Keeping	Keeping
4360	Under	4422	0.971	0.97	0.97	0.97	64	6	Keeping	Keeping
4360	Under	4402	0.976	0.98	0.97	0.98	64	6	Removing	Keeping
1913	Under	4078	0.981	0.97	0.99	0.98	64	6	Removing	Removing

Full field name for short words.

- PreP : Pre-processing
- Sen : Sentences
- Acc : Accuracy
- BatchS : Batch size
- Time : Training time
- EmptySen: Empty Sentences

Figure 9: Sample result after evaluation

8 Conclusion, Extensions and Issues

8.1 Conclusions

1. It will take more time to train less size of each batch.
2. Not much different in over-sampling and under-sampling method for dataset.
3. Removing empty sentence or stopwords will improve the result a little bit, that means we should remove those sentences in a large dataset..

8.2 Extensions

1. Initialize the embeddings with pre-trained word2vec vectors.
2. Using cross-validation for input dataset.

8.3 Currently issues in development

1. Word2Vec too heavy to load –¿ 40 min to run and crash PC.
2. Issue running only on Linux for prediction and accuracy without support on Pycharm and Window.
3. Tensorflow can not run with Python 3.7 only with 3.6 and lower.
4. Adding anaconda to manage tensorflow with PyCharm.
5. Tensorflow Allocation Memory: Allocation of 38535168 exceeds 10 percent of system memory. Then I need to reduce batch size or any argument to reduce calculation.

9 References

1. Implementing a CNN for Text Classification in TensorFlow.
2. Convolutional Neural Networks for Sentence Classification.
3. How do you apply SMOTE on text classification?.
4. NLP-progress.
5. ML Cheatsheet.