DEEP SEMANTIC FEATURE LEARNING FOR SOFTWARE DEFECT PREDICTION

Apurv Doddamani*, Vellore Institute of Technology Chennai, Tamil Nadu
Anushka Pandey†, Vellore Institute of Technology Chennai, Tamil Nadu
Lokesh Babu‡, Vellore Institute of Technology Chennai, Tamil Nadu

Abstract

Developers can use software defect prediction, which predicts problematic code sections, to detect flaws and prioritise their testing efforts. Traditional fault prediction algorithms frequently fail to account for semantic changes across programmes. The traditional model of prediction fails to adapt newer bugs . To create accurate prediction models, the ability to capture semantics in programmes is essential. In order to predict the new age we suggest using a deep learning mechanism which can properly give weight age to the semantic features and understand them in detail so that newer combating technologies can be developed We use a deep Keras Library to learn semantic features automatically using token vectors taken from the text file we provided. The model is then properly predicted using DBN and classifiers.

INTRODUCTION

The process of defect prediction identifies the bug during the automated testing process which prevents the development of faulty software modules. In this process, we have identified and used different data sets which are based on the earlier understanding of faulty data.

In this part of the code , an in-text document all sequences of words have their own index. Which helps for making a vector for the sentence, from which it counts each occurrence in the vocabulary. And at the end, the vector which we get has a length equal to the length of vocabulary and to each word present in it. The vector which we get is called a feature vector whose data type can be numerical or categorical. The feature vectors developed using the tokenizer model full form the basis of our analysis

To address the growing issues of bug prediction we rely upon the powerful machine learning algorithm Deep Belief Network. We then begin building the DBN model by using the RBM layers to construct the input of DBN. We then leverage the use of Deep Belief Network(DBN) for learning features, we need to tune three parameters, which are: 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the number of training iterations. After we finish building the model we plot the error plot w.r.t to the number of iterations. In the end , we conclude by using Logistic Regression Classifier to predict F1 Score and Accuracy

* apurv.sangmesh2019@vitstudent.ac.in
† anushka.pandey2019@vitstudent.ac.in
‡ kancharlalokesh.babu2019@vitstudent.ac.in

SEMANTIC FEATURE CLASSIFICATION

We begin semantic feature learning with importing text file.Then we construct a CSV file containing a sentence and label.Here we label Positive label as 1 and Negative Label as 0.This is where we try to differentiate the semantic features which will later play a great enabler for our work

Then we count the frequency of each word in each sentence and then in the entire database.This way we create a vocabulary for the whole thing.Then we try to vectorize then sentence by creating a vector of the words from the vocabulary.A feature vector is the product of this process. Each dimension of a feature vector can be a numeric or categorical feature, such as a building's height, the price of a stock, or, in our instance, the number of words in a lexicon.

Choosing the data set

```
import pandas as pd

filepath_dict = {'yelp':   'yelp_labelled.txt',
                 'amazon': 'amazon_cells_labelled.txt',
                 'imdb':   'imdb_labelled.txt'}

df_list = []
for source, filepath in filepath_dict.items():
    df = pd.read_csv(filepath, names=['sentence', 'label'], sep='\t')
    df['source'] = source  # Add another column filled with the source name
    df_list.append(df)

df = pd.concat(df_list)
print(df.iloc[0])
```

We here have chosen the text file from which we are going to extract the feature

BUILDING THE MODEL

We here build a model using the text file to build the token vector without using the KERAS library

```
from sklearn.model_selection import train_test_split
df_yelp = df[df['source'] == 'yelp']

sentences = df_yelp['sentence'].values
y = df_yelp['label'].values

sentences_train, sentences_test, y_train, y_test = train_test_split(sentences, y, test_size=0.25, random_state=1000)
```

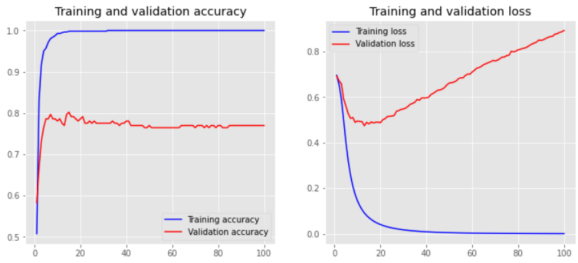
```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(sentences_train)
X_train = vectorizer.transform(sentences_train)
X_test  = vectorizer.transform(sentences_test)
X_train
```

ACCURACY

```
for source in df['source'].unique():
    df_source = df[df['source'] == source]
    sentences = df_source['sentence'].values
    y = df_source['label'].values

    sentences_train, sentences_test, y_train, y_test = train_test_split(
        sentences, y, test_size=0.25, random_state=1000)

    vectorizer = CountVectorizer()
    vectorizer.fit(sentences_train)
    X_train = vectorizer.transform(sentences_train)
    X_test  = vectorizer.transform(sentences_test)

    classifier = LogisticRegression()
    classifier.fit(X_train, y_train)
    score = classifier.score(X_test, y_test)
    print('Accuracy for {} data: {:.4f}'.format(source, score))
```

```
Accuracy for yelp data: 0.7960
Accuracy for amazon data: 0.7960
Accuracy for imdb data: 0.7487
```

Plots:



BUILDING THE MODEL USING THE KERAS LIBRARY

The KERAS model is a linear stack of layers, where you can use the large variety of available layers in Keras. The most common layer is the Dense layer which is your regular densely connected neural network layer with all the weights and biases that you are already familiar with.

```
from keras.models import Sequential
from keras import layers

input_dim = X_train.shape[1]  # Number of features

model = Sequential()
model.add(layers.Dense(10, input_dim=input_dim, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

We use the loss function and the accuracy to accuracy for training and testing data set.

```
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
Training Accuracy: 1.0000
Testing Accuracy:  0.7701
```

PLOTTING THE GRAPH FOR ACCURACY AND LOSS

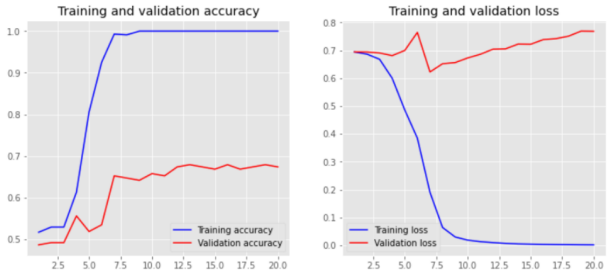
Defining plotting function:

```
def plot_history(history):
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    x = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(x, acc, 'b', label='Training accuracy')
    plt.plot(x, val_acc, 'r', label='Validation accuracy')
    plt.title('Training and validation accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(x, loss, 'b', label='Training loss')
    plt.plot(x, val_loss, 'r', label='Validation loss')
    plt.title('Training and validation loss')
    plt.legend()
```

KERAS EMBEDDING LAYER

Keras is a text classification library that is used to embed different layers into the neural network Integer encoding of the input data is required, with each word represented by a separate number. This data preparation phase can be carried out using Keras Tokenizer API. Starting with random weights, the Embedding layer learns an embedding for each word in the training data set.

```
Training Accuracy: 1.0000
Testing Accuracy:  0.6738
```



CONVOLUTIONAL NEURAL NETWORKS (CNN)

Convolutional neural networks, or convents, are one of the most interesting breakthroughs in recent years in machine learning. By extracting features from photos and using them in neural networks, they have revolutionized image categorization and computer vision. They have the same qualities that make them valuable for image processing that also make them useful for sequence processing. A CNN can be thought of as a customized neural network capable of detecting specific patterns.

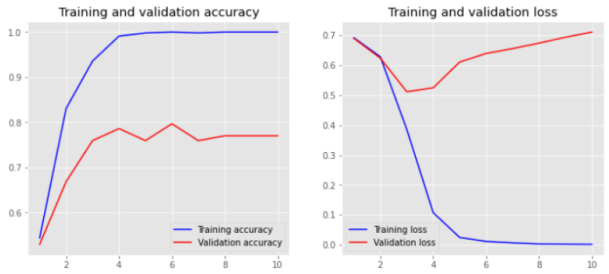
When working with sequential data, such as text, one-dimensional convolutions are used, but the concept and application remain the same. You'll still want to look for patterns in the sequence that become more complicated as more convolutional layers are added.

```
embedding_dim = 100

model = Sequential()
model.add(layers.Embedding(vocab_size, embedding_dim, input_length=maxlen))
model.add(layers.Conv1D(128, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 100)          257500

conv1d (Conv1D)              (None, 96, 128)           64128

global_max_pooling1d (Globa  (None, 128)               0
lMaxPooling1D)

dense_2 (Dense)              (None, 10)                1290

dense_3 (Dense)              (None, 1)                 11

=================================================================
Total params: 322,929
Trainable params: 322,929
Non-trainable params: 0
_________________________________________________________________
```

```
history = model.fit(X_train, y_train,
                    epochs=10,
                    verbose=False,
                    validation_data=(X_test, y_test),
                    batch_size=10)
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
plot_history(history)
```

```
Training Accuracy: 1.0000
Testing Accuracy:  0.7701
```



DEEP BELIEF NETWORK

Normalization Function Code Min-Max Re-scaling: Shifting and squeezing a distribution to fit on a scale between 0 and 1. Min-Max Re-scaling is useful for comparing distributions with different scales or different shapes.

```
def normalize(x):
  x = x.astype(float)
  min = np.min(x)
  max = np.max(x)
  return (x - min)/(max-min)
```

Before Normalizing data set:

```
[ ] df.head()
```

	transactionid	commitdate	ns	nm	nf	entropy	la	ld	lt	fix	ndev	pd	npt	exp	rexp	sexp	bug
0	3	2001/12/12 17:41	1	1	3	0.579380	0.093820	0.000000	480.666667	1	14	596	0.666667	143	133.50	129	1
1	7	1999/10/12 12:57	1	1	1	0.000000	0.000000	0.000000	398.000000	1	1	0	1.000000	140	140.00	137	1
2	8	2002/5/15 16:55	3	3	52	0.739279	0.183477	0.208913	283.519231	0	23	15836	0.750000	984	818.65	978	0
3	9	2002/1/21 15:37	1	1	8	0.685328	0.016039	0.012880	514.375000	1	21	1281	1.000000	579	479.25	550	0
4	10	2001/12/19 16:44	2	2	38	0.769776	0.091829	0.072746	366.815789	1	21	6565	0.763158	413	313.25	405	0

After Normalizing data set:

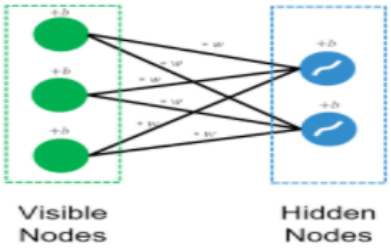
Splitting and Normalizing dataframe

```
[ ] #split df
    train_X = df.iloc[:,:-1].apply(func=normalize, axis=0)
    train_Y = df.iloc[:,-1]
```

```
[ ] # df=df.drop(['transactionid'], axis=1)
    print(df.head())
    df.shape
```

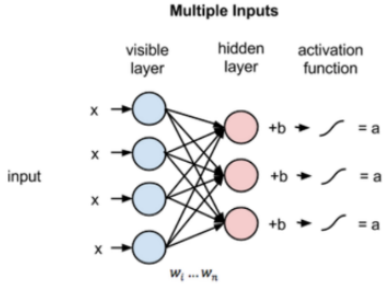
```
   ns  nm  nf   entropy        la  ...       npt  exp    rexp  sexp  bug
0   1   1   3  0.579380  0.093620  ...  0.666667  143  133.50   129    1
1   1   1   1  0.000000  0.000000  ...  1.000000  140  140.00   137    1
2   3   3  52  0.739279  0.183477  ...  0.750000  984  818.65   978    0
3   1   1   8  0.685328  0.016039  ...  1.000000  579  479.25   550    0
4   2   2  38  0.769776  0.091829  ...  0.763158  413  313.25   405    0

[5 rows x 15 columns]
(4620, 15)
```

RESTRICTED BOLTZMANN MACHINE(RBM)



The Restricted Boltzmann Machine is an algorithm for unsupervised learning. It has a visible layer of neurons that accepts input data and multiplies it by specific weights so that it can be combined with a bias value at the hidden layer to generate output The hidden layer neuron's output value is then multiplied with the same weights, and the bias of the visible layer is then added to regenerate input.Backward pass is the terminology used for this procedure. The rebuilt input will be stacked against the original values for comparison. The above process continues till the rebuilt input matches the original value

In RBM we have calculated activation value using formula:- activation f((weight w * input x) + bias b) = output a



EVALUATION METRICS

$$Precision = \frac{true\ positive}{true\ positive + false\ positive}$$

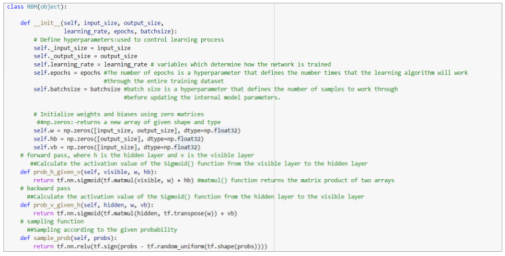
$$Recall = \frac{true\ positive}{true\ positive + false\ negative}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

The F-score, also known as the F1-score, is a metric for how accurate a model is on a given dataset. The F-score,

which is defined as the harmonic mean of the model's precision and recall, is a technique of combining the model's precision and recall

RBM IMPLEMENTATION CODE:-

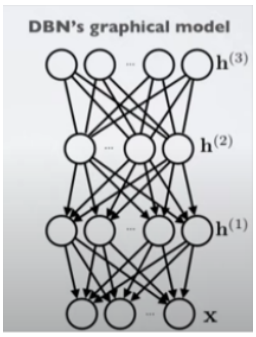


```
class RBM(object):

    def __init__(self, input_size, output_size,
                 learning_rate, epochs, batchsize):
        # Define hyperparameters:used to control learning process
        self._input_size = input_size
        self._output_size = output_size
        self.learning_rate = learning_rate # variables which determine how the network is trained
        self.epochs = epochs #The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work
                             #through the entire training dataset
        self.batchsize = batchsize #batch size is a hyperparameter that defines the number of samples to work through
                               #before updating the internal model parameters.

        # Initialize weights and biases using zero matrices
         ##np.zeros:-returns a new array of given shape and type
        self.w = np.zeros([input_size, output_size], dtype=np.float32)
        self.hb = np.zeros([output_size], dtype=np.float32)
        self.vb = np.zeros([input_size], dtype=np.float32)
    # forward pass, where h is the hidden layer and v is the visible layer
      ##calculate the activation value of the Sigmoid() function from the visible layer to the hidden layer
    def prob_h_given_v(self, visible, w, hb):
        return tf.nn.sigmoid(tf.matmul(visible, w) + hb) #matmul() function returns the matrix product of two arrays
    # backward pass
      ##calculate the activation value of the Sigmoid() function from the hidden layer to the visible layer
    def prob_v_given_h(self, hidden, w, vb):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
    # sampling function
      ##sampling according to the given probability
    def sample_prob(self, probs):
        return tf.nn.relu(tf.sign(probs - tf.random_uniform(tf.shape(probs))))
```

```
def train(self, X):
    #A placeholder is simply a variable that we will assign data to at a later date
    _w = tf.placeholder(tf.float32, [self._input_size, self._output_size])
    _hb = tf.placeholder(tf.float32, [self._output_size])
    _vb = tf.placeholder(tf.float32, [self._input_size])

    prv_w = np.zeros([self._input_size, self._output_size], dtype=np.float32)
    prv_hb = np.zeros([self._output_size], dtype=np.float32)
    prv_vb = np.zeros([self._input_size], dtype=np.float32)

    cur_w = np.zeros([self._input_size, self._output_size], dtype=np.float32)
    cur_hb = np.zeros([self._output_size], dtype=np.float32)
    cur_vb = np.zeros([self._input_size], dtype=np.float32)

    ##check
    v0 = tf.placeholder(tf.float32, [None, self._input_size])
    h0 = self.sample_prob(self.prob_h_given_v(v0, _w, _hb))
    v1 = self.sample_prob(self.prob_v_given_h(h0, _w, _vb))
    h1 = self.prob_h_given_v(v1, _w, _hb)
    #To update the weights, we perform constrastive divergence.
    positive_grad = tf.matmul(tf.transpose(v0), h0)
    negative_grad = tf.matmul(tf.transpose(v1), h1)
```

```
# Calculate and update each parameter
update_w = _w + self.learning_rate * (positive_grad - negative_grad) / tf.to_float(tf.shape(v0)[0])
update_vb = _vb +  self.learning_rate * tf.reduce_mean(v0 - v1, 0)
update_hb = _hb +  self.learning_rate * tf.reduce_mean(h0 - h1, 0)
# We also define the error as the MSE
err = tf.reduce_mean(tf.square(v0 - v1))

error_list = []

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(self.epochs):
        for start, end in zip(range(0, len(X),  self.batchsize),range(self.batchsize,len(X), self.batchsize)):
            batch = X[start:end]
            cur_w = sess.run(update_w, feed_dict={v0: batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            cur_hb = sess.run(update_hb, feed_dict={v0: batch,  _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            cur_vb = sess.run(update_vb, feed_dict={v0: batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            prv_w = cur_w
            prv_hb = cur_hb
            prv_vb = cur_vb
        error = sess.run(err, feed_dict={v0: X, _w: cur_w, _vb: cur_vb, _hb: cur_hb})
```

DBN



•The Idea of Pre-Training came from work on Deep Belief Network.

- It is a generative model that mixes undirected and directed connections between variables
- Deep Belief Networks consist of multiple layers with values, wherein there is a relation between the layers but not the values
- Deep Belief Networks are composed of unsupervised networks like RBMs. In this the invisible layer of each sub network is the visible layer of the next.
- Deep-belief networks are used to recognize, cluster and generate images, video sequences and motion-capture data

A Deep Belief Network (DBN) is a multi-layer generative graphical model.

The top layer of a DBN has bidirectional connections (RBM-type connections), while the bottom layers only have top-down connections. Layer-by-layer pre-training is used to train them. Pre-training is done by training the network from the bottom up, treating the first two layers as an RBM and training them, then treating the second and third levels as another RBM and training them.

DBN CODE IMPLEMENTATION

```
class DBN(object):
    def __init__(self, original_input_size, input_size, output_size,
                 learning_rate, epochs, batchsize, rbmOne, rbmTwo, rbmThree):
        # Define hyperparameters
        self._original_input_size = original_input_size
        self._input_size = input_size
        self._output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batchsize = batchsize
        self.rbmOne = rbmOne
        self.rbmTwo = rbmTwo
        self.rbmThree = rbmThree

        self.w = np.zeros([input_size, output_size], "float")
        self.hb = np.zeros([output_size], "float")
        self.vb = np.zeros([input_size], "float")

    def prob_h_given_v(self, visible, w, hb):
        return tf.nn.sigmoid(tf.matmul(visible, w) + hb)

    def prob_v_given_h(self, hidden, w, vb):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
```

LOGISTIC REGRESSION

Logistic regression is a classification model rather than a regression model. It is a simple and more efficient method for binary and linear classification problems. It is a classification model, which is very easy to realize and achieves very good performance with linearly separable classes. Here we are using logistic regression by passing values of dbn output and original y output in the gradient descent function.

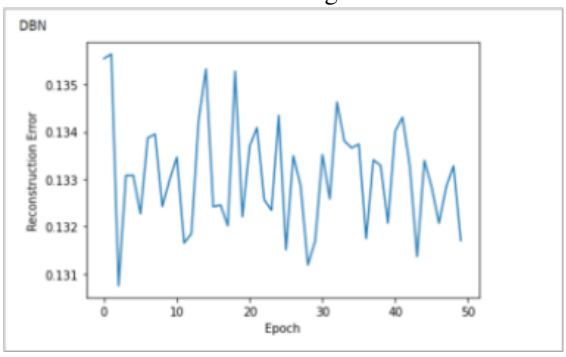
We are calculating error till maximum iterations using hypothesis function and error function defined. Here we are defining a sigmoid function for calculating hypothesis of output generated using dbn and original y target variable.

```
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def hypothesis(X, theta) :
    #returns the dot product of vectors X and theta
    return sigmoid(np.dot(X, theta))

def error(X,y,theta):
    hi = hypothesis(X,theta)
    error= -1* np.mean ( y * np.log(hi) + ( ( 1 - y ) * (np.log( 1 - hi )) ) )

    return error
```

```
def gradient(X,y,theta):
    hi = hypothesis(X,theta)
    #X.T :transpose of X
    grad = np.dot(X.T,(y-hi))
    m=X.shape[0]

    return grad/m
```

EXPERIMENTAL RESULTS

1.For dataset1:bugzilla.csv

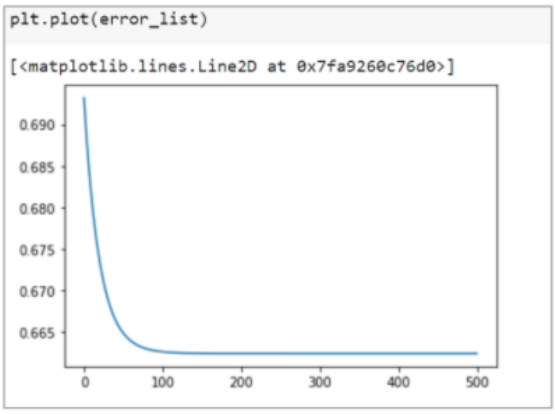


Accuracy:-

```
train_acc= accuracy(Y_Train,XT_preds)
print(train_acc)

62.324999999999996
```

Regression classifier:-

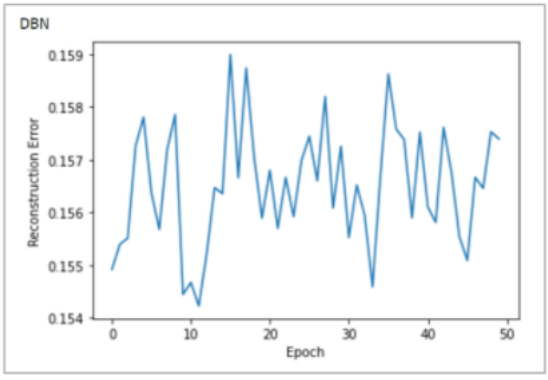
```
plt.plot(error_list)

[<matplotlib.lines.Line2D at 0x7fa9260c76d0>]
```



F1 Score:-

```
print(f1_score(Y_Train,XT_preds,average='micro'))

0.62325
```

2.For dataset2: columba.csv

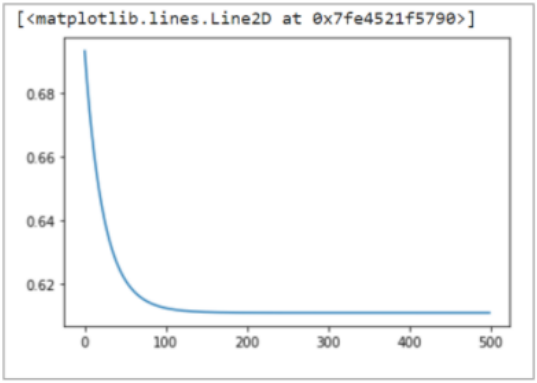


Accuracy:-

```
train_acc= accuracy(Y_Train,XT_preds)
print(train_acc)

70.0
```

Regression classifier:-

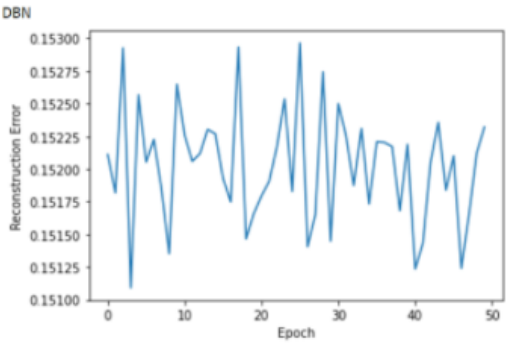
```
[<matplotlib.lines.Line2D at 0x7fe4521f5790>]
```



F1 Score:-

```
print(f1_score(Y_Train,XT_preds,average='micro'))

0.7
```

3.For dataset3: postgres.csv

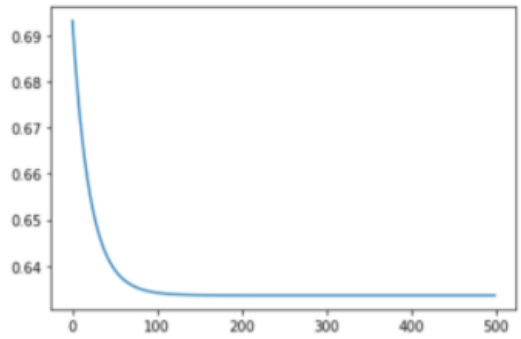


Accuracy:-

```
train_acc= accuracy(Y_Train,XT_preds)
print(train_acc)

67.07499999999999
```

Regression classifier:-

```
[<matplotlib.lines.Line2D at 0x7fc1213a0a10>]
```



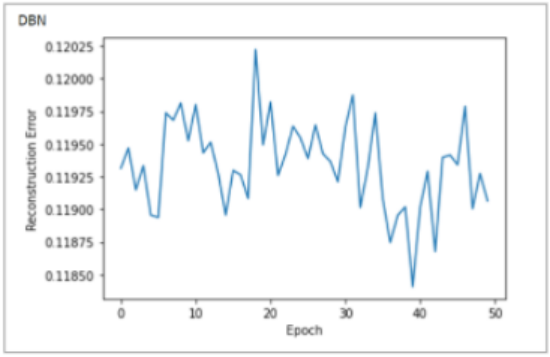
F1

Score:-

```
print(f1_score(Y_Train,XT_preds,average='micro'))
```

```
0.67075
```

4.For dataset4: jdt.csv



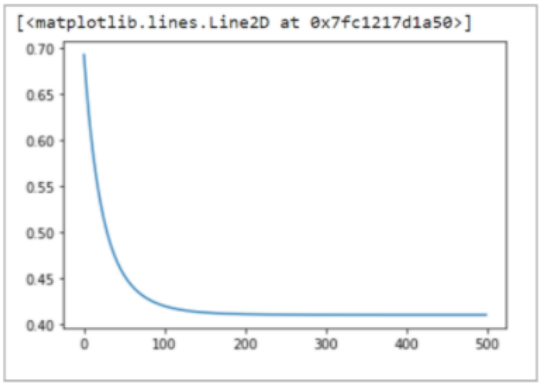
Accuracy:-

```
train_acc= accuracy(Y_Train,XT_preds)
print(train_acc)
```

```
85.7
```

Regression classifier:-

```
[<matplotlib.lines.Line2D at 0x7fc1217d1a50>]
```



F1

Score:-

```
print(f1_score(Y_Train,XT_preds,average='micro'))
```

```
0.857
```

RESEARCH GAP

In the given paper , we can see that for the logistic classifier the average f1 score for the different data sets is 0.49 for the PROMISE data set.

The text classification using keras library with different dataset produces good semantic feature recognition with accuracy of 0.78 as compared to the paper.

There has been no mention of error calculation and low accuracy of the model which we see as the research gap which the authors didn't explain decisively. In our case we have used data set Bugzilla, Columba, jdt, Postgres for the calculation of Accuracy and F1 Score.

Dataset used	Bugzilla	Columba	jdt	Postgres
Accuracy	62.325	70.0	85.7	67.0745
F1 Score	0.62325	0.7	0.857	0.67075

CONCLUSIONS AND FUTURE WORK

Finally, we'd like to mention that we used Text Classification and the DBN model with classifier. We hope to enhance our DBN-based methodology in the future to build a full-fledged model for semantic features learning, which will aid in anticipating problematic methods in software projects. It could be beneficial to use the defect prediction result to help with other software development and maintenance procedures. For example, Quality Assurance teams have utilised software defect prediction to help prioritise test cases, improve static bug finders, and so on. We intend to investigate how defect prediction could be used to improve risk management, quality control, and project planning.

ACKNOWLEDGEMENTS

This work has been supported by our professor Dr Shivani Gupta.The authors thank the professors for their feedback which helped improve this paper.

REFERENCES

https://www.sciencedirect.com/topics/engineering/deep-belief-network

https://monkeylearn.com/text-classification

https://github.com/kavetinaveen/Deep-Learning-for-Semantic-Text-Matching

https://github.com/felixSchober/Defect-Prediction

https://www.cs.purdue.edu/homes/lintan/publications/deeplearn-tse18.pdf

https://ieeexplore.ieee.org/abstract/
document/7018910