IMPROVING PERFORMANCE OF MACHINE LEARNING MODELS

MIS 800 Special problems in MIS)

Abstract

Final research paper focusing on methods to improve performance of ML models in NLP via data cleaning and hyperparameter tuning

Shreyas Menon

CWID: 10426882

Mentor

Professor Rong (Emily) Liu

Improving performance of machine learning models

Motivation

In the current era of automation, due to the rise in use of machine learning applications and its potential demand in the future, a lot of organizations have started providing libraries for doing almost everything in machine learning. Starting from pulling data from a csv file to applying sophisticated algorithms over it, all the required libraries, APIs and code blocks are available. We're even entering the age where machine learning suites can be used to build production ready machine learning applications like H20's driverless AI and Microsoft Azure machine learning.

But few of the most sought after skills that differentiates a Data scientist from an application developer are

1. The ability to Identify the nature of problem to be solved – for example if it's a classification or regression problem, or simply statistical analysis is required.
2. Ability to apply the most suitable machine learning algorithms to that use case
3. Ability to 'tune' the parameters and hyperparameters to extract the optimum performance

The above skills are truly influential for the final outcome and many industry experts believe that It's near to impossible to take these skills away from a Data scientist and automate them.

This research paper addresses the above problems of machine learning in Natural language processing (NLP) with the help of a real-life use case. The audience for this paper can either be an application developers seeking examples to leverage the mentioned 3 skills or someone with an interest in machine learning. The overall intention of this paper is to showcase how the performance of machine learning algorithms can be improved.

The code repository for this report can be found at

https://github.com/shreyasmenon/improving_machinelearning_performance

Introduction

Consider a Telecom company named 'Stevens Telecommunications' (referred to as ST hereon) which provides internet, cable and phone services to its subscribers. The subscriber or customers of ST can have subscriptions to either one, two or all of the services. Due to its huge customer base, ST receives a lot of support requests via different channels and the volume is ever growing. The operations department head in the contact center wishes to analyze these incoming requests

and wishes to use the data to make better business decision. The head wishes to leverage machine learning to identify the intentions of these customer inquiries.

In a nutshell, the problem statement is 'To Identify the intention of incoming support requests using machine learning' **TO**

a. Automatically tag incoming request and direct them to suitable departments. Example billing support requests goes to billing department.
b. If a ticket is raised after the request, apply appropriate tags to them.
c. Aggregate tickets using the above tags and generate timely reports and get insights

Architecture

Step 1: Identifying the problem

The head wishes to identify the intent and categorize them. Hence, it's a straightforward classification problem

Step 2: Choice of machine learning algorithm

Considering the need to classify text from support requests, coming from social media, emails or voice transcripts, it may involve dealing with highly dimensional and unstructured data.

Supervised approach

ST's previous attempt of unsupervised learning using Topic modelling didn't yield expected performance and therefor the other approach was manually labelling a subset of the data for a supervised learning.

One of the better performing classification algorithms for high dimensional data is 'Support Vector machines' and the plan was to use the same with highest advantage. For benchmarking purposes a more traditional Naïve Bayes' algorithm was be used and for comparing performances, Deep Neural networks were used.

Step 3: Improving performance

The algorithms were initialized to random input variables and based upon the performance, the tuning was performed.

Implementation

To establish a ground truth and training a supervised model, Analysts from ST took a subset of the data available and started labeling them manually. The team ended up labelling around 5000 samples.

a. Cleaning the data

To obtain optimal performance, one of the primary steps in NLP is to clean the data. Cleaning of data involves the following processes:

i. Tokenizing

Tokenizing helps in striping punctuations and other characters in text that are of less importance. Based upon the use-case, nltk provided a variety of tokenizers like word tokenizer, punct tokenizer, Twitter tokenizer,etc.

For the current use case, the basic word tokenizer is used to tokenize the text.

ii. Removing Stop words

Stop words are frequently occurring words like 'a',' the','to',etc. which has less impact on the classification model's performance. A complete list of stopwords from nltk can be found here

iii. Lemmatizing

Lemmatizing is the process of extracting 'Lemma' or root of the word tokens. It is an example of normalizing the words in the text. The ultimate goal of lemmatizing is to establish a common base for word tokens.

For example: 'walked', 'walks', 'walking' becomes 'walk' and they are one feature instead of 3.

The cleaning of the data is done by the following function

```
    def tokenize_lemmatize(df_queries, use_stopwords = True
extract_entities = True):
```

b. Building the model

The first step for training the supervised model is to split the samples in train and test. In all, the total labelled samples are 5000 which are split into 80% training and 20% testing

```
X_train, X_test, y_train, y_test =
train_test_split(df['query_request'], df['intent'], test_size=0.2,
random_state=0)
```

The X_train and y_train are fed to all models for training

a. Naïve Bayes classification

Naïve Bayes is classic classification technique which is based upon Bayes Theorem. It is one the simplest and effective supervised classification algorithm. Naïve Bayes calculates prior and posterior probabilities to classify a sample.

Gaussian Naïve Bayes distribution is used and it is fitted on to training samples as

```
nb_model = GaussianNB()
nb_model.fit(X_train,y_train)
```

b. Support Vector machines

Support vector machines is one of the better classifiers while dealing with high dimensional and non-linear data samples. Support vector machine for classification is termed as SVC – Support vector classifier. SVM algorithm basically adapts a hyperplane in multidimensional space to separate different classes.

SVC model from Sklearn is fitted for training sample as

```
from sklearn.svm import SVC
svc_model = SVC(gamma='auto')
svc_model.fit(X_train, y_train)
predicted= svc_model.predict(X_test)
```

c. Deep neural networks

Deep neural networks are more advanced algorithms with variety of applications like classification, image recognition, computer vision, etc. It works mostly in the form of

'layers' where each layer will have set of inputs, outputs, activation functions, weights and biases. It works through number of iterations where the weights and biases adjust and learn themselves to predict the samples with least error. Gradient descent is one of the widely-used learning methods.

A sequential neural network model with 1 input layer, 2 hidden layer and 1 output layer, making it a 4-layer Neural network. Neural network from tensorflow's keras was used

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential

nn_model = Sequential()
nn_model.add(layers.Embedding(input_dim=input_dim,
output_dim=embedding_dim, input_length=maxlen))
nn_model.add(layers.GlobalMaxPool1D())
nn_model.add(layers.Dense(512, activation='relu'))
nn_model.add(layers.Dense(9, activation='sigmoid'))
nn_model.compile(loss='categorical_crossentropy',metrics=['accuracy'
])
history = nn_model.fit(NN_X_train, NN_y_train, epochs=50,
verbose=True, validation_data=(NN_X_test, NN_y_test), batch_size=10)
loss, accuracy = nn_model.evaluate(NN_X_train, NN_y_train,
verbose=True)
```

Most often, for classification applications of Neural nets, the activation function used in the output layer will be sigmoid.

c. Testing the performance

The performance of each model is tested using sklearn's classification report. The performance is tested by comparing the predictions of each model and the actual test samples

```
print(classification_report(y_test,predicted))
```

Performance for Naïve Bayes Classifier

	precision	recall	f1-score	support
account related	0.00	0.00	0.00	57
appointment related	0.00	0.00	0.00	49
billing	0.00	0.00	0.00	205
internet	0.33	1.00	0.49	334
phone	0.00	0.00	0.00	75
sales	0.00	0.00	0.00	112
video	0.00	0.00	0.00	190
avg / total	0.11	0.33	0.16	1022

Naïve Bayes doesn't perform well with the given dataset

Performance for SVM classifier

	precision	recall	f1-score	support
account related	0.50	0.32	0.39	57
appointment related	0.67	0.04	0.08	49
billing	0.71	0.54	0.61	205
internet	0.55	0.94	0.69	334
phone	0.97	0.39	0.55	75
sales	0.80	0.68	0.73	112
video	0.84	0.58	0.69	190
avg / total	0.70	0.65	0.62	1022

SVC performance is impressive and it's certain that there's scope for better performance.

Performance of Neural Networks classifier

```
Train on 4084 samples, validate on 1022 samples
4084/4084 [==============================] - 2s 402us/sample - loss: 1.8371 - acc: 0.3210 - val_loss:
1.7160 - val_acc: 0.3268
4084/4084 [==============================] - 0s 33us/sample - loss: 1.7151 - acc: 0.3210
Training Accuracy: 0.3210
1022/1022 [==============================] - 0s 37us/sample - loss: 1.7160 - acc: 0.3268
Testing Accuracy:  0.3268
```

Neural networks classifier doesn't perform well as expected. However, the all-important parameter epoch is not set while training.

Improving the performance of the models and their evaluation

To make the classifier useful in real life, it's still important to extract higher performances from it. Hence, SVC and Neural network classifiers were considered one by one and their hyperparameters were tuned for highest performance.

Better cleaning strategy

a. **Having correct list of stopwords**

NLTK's list of stopwords can be found here. After examining the data and nltk's list of stopwords, it was observed that nltk's list was extensive and it was required to be trimmed per our use case.

For example: nltk includes 'not' and 'doens't' in its list of stopwords. However, excluding not from our query request text completely flips the meaning of the sentence.

'internet not working' becomes *'internet working'*

'phone doesn't work' becomes '*phone work*'

Hence it was necessary to explicitly remove these stopwords to maintain the sanity of the classification.

```
stop_words = stopwords.words('english')
stop_words.remove('not')
stop_words.remove("doesn't")
```

b. **Removal of rubbish samples from training set and dropping those**

Once the samples were tokenized and lemmatized, a peek into the filtered set revealed that the many samples were useless and could be removed for better training the model.

For example "welcome","good night","hello",etc all these pleasantries were left after lemmatization and had little value in them, for being used for any classification.

A decent overview of the filtered set lead to formation of this exhaustive list of words

```
list_null_values =['any update','anything','anything else','but
better yet','earl hedin','fine','good morning','good night','got
it thanks','have a good night','hello','help','hey', 'hi','i did
already','i don','sorry i tried that as well','issue fixed','is
```

```
that a no?','no','nope','not working right
now','ok','sure','yes','yea', 'welcome','yw','thanks','thank']
```

And later, samples equaling to all these words were directly replaced by 'na' and deleted later on as

```
df['query_request'] = df['query_request'].apply(lambda x: np.nan
if x in list_null_values else x)

df.dropna(inplace = True)
```

Now the data is much more cleaner and the algorithms can be trained a lot better

c. Extracting entities from unstructured data

Now that the data was scrubbed enough, the next challenge was to let the machine learning algorithms identify patterns in the data better, so that it would classify them better. Reflecting from the performance of the SVM model before, one of the classes that was performing bad was '*account_related*"

	precision	recall	f1-score	support
account related	0.50	0.32	0.39	57
appointment related	0.67	0.04	0.08	49
billing	0.71	0.54	0.61	205
internet	0.55	0.94	0.69	334
phone	0.97	0.39	0.55	75
sales	0.80	0.68	0.73	112
video	0.84	0.58	0.69	190
avg / total	0.70	0.65	0.62	1022

The recall and f1 score are very less than acceptable and had scope of improvement.

A peek into the training samples for this class revealed that many samples from this class had information like

'John smith account number 123-123-12 '

'Kristina jolie phone number 0000-1111-123'

'Arya Stark 1 Castle point terrace, Hoboken, New Jersey'

For human eye, this data was certainly *account* information of the subscribers or account holders. However, for the machine, it was a list of unrelated text with no common pattern. Therefore, it was necessary to structure these into appropriate entities to improve the performance. For this purpose, 'regular expressions' were used.

Regular expressions are a sequence of characters that identifies patterns from a long string or text. In Python, a common package for building and executing regular expressions is 'regex'. Regular expressions for observing 3 patterns were built

1. Account numbers

```
r"(\d{5}[-\.\s]??\d{6}[-\.\s]??\d{2}[-\.\s]??\d{1}|\(\d{5}\)\s*\d{6}[-
\.\s]??\d{2}[-\.\s]??\d{1})"
```

2. Phone numbers

```
r'(\d{3}[-\.\s]??\d{3}[-\.\s]??\d{4}|\(\d{3}\)\s*\d{3}[-
\.\s]??\d{4}|\d{3}[-\.\s]??\d{4})'
```

3. Addresses

```
r'([0-9]{1,4}
[\w]{1,20}(.*)(st|street|road|rd|blvd|boulevard|ave|avenue|ct|drive|dr|
lane|ace|way)(.{0,20})(nj|ny|ct|new                    jersey|new
york|connecticut)[\s]{0,}[\d]{0,5})'
```

Therefore, in a data sample, if account information in any of the above patterns was observed, it would be replaced by keywords '`accountnumberprovided`' '`phonenumberprovied`' and '`addressprovided`'. For example,

```
meta_response = re.sub(phone_pattern,' phoneprovided ',meta_response)
```

Extracting names

Extracting names was still a challenge as regular expressions don't provide a methodology for that. The first option was to extract part of speech tag 'NNP' from upenn tagset, however it wasn't much useful and it couldn't detect names with high confidence. Another approach used was spacy's named entity recognition

Spacy's Named Entity Recognition

Spacy provides intelligent and convenient way to identify entities from raw text. Complex entities like Human names (PERSON), Organization/company names (ORG) and currency information (MONEY) can be easily extracted.

Hence to identify human names

```
list_human_names = [ entity.text for entity in spacy_doc.ents if entity.label_ == 'PERSON' if str(entity.text).isalpha() ]
```

and later

```
  if len(list_human_names) > 0:
        for name in list_human_names:
                  meta_response      =      meta_response.replace(name,' nameprovided ')
```

In all, a sentence before entity extraction would be like

'Arya Stark 1 Castle point terrace, Hoboken, New Jersey phone number 0001111234 and account number 123-12311'

And after entity extraction would be as

'nameprovided addressprovided phone number phonenumberprovided and account number accountnumberprovided'

Using this structured entity extraction, a machine would identify and classify such patterns better.

To add flexibility, a default argument was added to tokenize_lemmatize function

```
def tokenize_lemmatize(df_queries, use_stopwords = True , extract_entities = True):
```

d. Preserving useful information using TweetTokenizer

NLTK provides an interesting tokenizer TweetTokenizer, that preserves useful information like #tags and @handles, by stripping the punctuation and keeping the text

```
tknzr = TweetTokenizer(strip_handles=True)
```

Performance improvement for Support vector machine classifier

To start, consider the previous SVC performance

	precision	recall	f1-score	support
account related	0.50	0.32	0.39	57
appointment related	0.67	0.04	0.08	49
billing	0.71	0.54	0.61	205
internet	0.55	0.94	0.69	334
phone	0.97	0.39	0.55	75
sales	0.80	0.68	0.73	112
video	0.84	0.58	0.69	190
avg / total	0.70	0.65	0.62	1022

In addition to the data cleaning strategy mentioned above, following were the strategies used for improving SVC classification's performance

a. Use of 'pipeline' from sklearn for countvectorizing, tf-idf and classification

Sklearn provides a very convenient methodology of using a 'pipeline' to build a machine learning model. As the name states, 'pipeline' applies a sequence of transformations and produces a final estimator. Instead of having 3 different steps in building the model, the same can be built in a single line of code

```
mdl = Pipeline([('vectorizer', CountVectorizer()),('tfidf',
TfidfTransformer()),('clf', SVC())])
```

This way, we don't need to handle intermediate complex numerical arrays and the pipeline will handle them by itself. Also, once the model is trained, for predicting a textual sample, we don't need to apply preprocessing steps to the testing sample. Instead we can just provide the sample and the model would return the predicted class.

In addition to being cleaner and simpler, the main advantage of using methodology is the final estimator can be directly fed into GridSearch CV and GridSearchCV can run all three operations using a single estimator and therefore we can shift our focus to parameters and hyperparameters.

b. Use of GridSearch CV

GridSearchCV is a convenient method of executing an exhaustive search to find the best parameters. It applies all combinations of the provided parameters and builds a model, which then outputs the best combination to get the maximum value of the given performance metric.

Preprocessing parameters

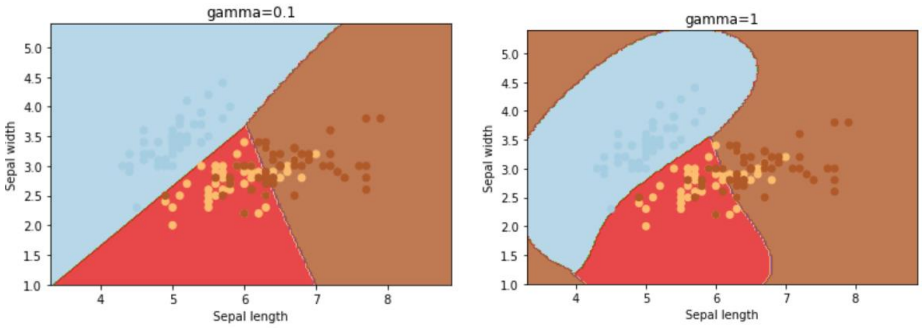
i. Vectorizer ngram range

It is used during computing the countvectorizer. It is basically a range of min and max values of n-grams that are to be extracted.

For SVC, parameters that are of relevance are

i. Gamma

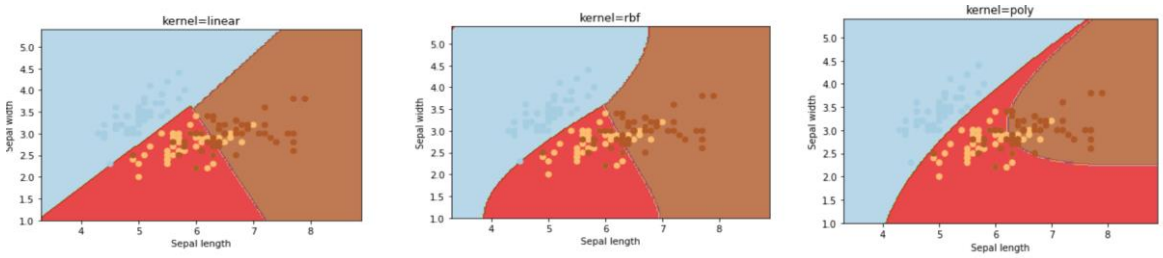
Gamma deals with nonlinear classification. A small value of gamma gives more linear and sharp classification and a larger value gives a smoothed or nonlinear classification



https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769

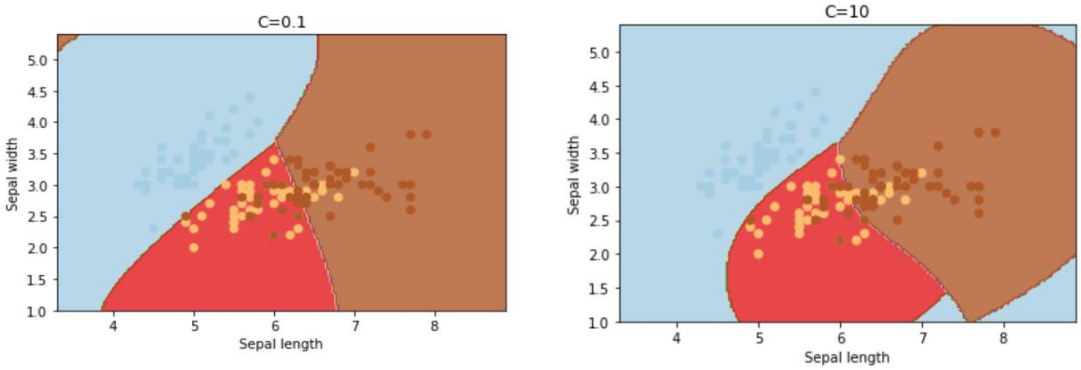
ii. Kernel

As SVM uses a hyperplane to classify and separate samples in space, the choice of the kernel denotes the 'type' of the hyperplane that can be used. The kernel type that are available are 'linear' , 'rbf' and 'poly'
Linear – for linear hyperplanes
Rbf and ploy – for nonlinear hyperplanes



https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769

iii. C
C is the tradeoff for misclassification. A very high C value may result in overfitting of the data



In all, the parameter array to be fed to grid search CV looks like

```
parameters =
{'clf__C':[1,10,100,1000],'clf__gamma':[1,0.1,0.001,0.0001],
'clf__kernel':['linear','rbf','poly'] , 'vectorizer__ngram_range': [(1,
1), (1, 2),(2,2)]}
```

A target metric can be specified for which the Gridsearch should obtain the maximum value. The target metric used here was 'accuracy'.

After providing number of crossvalidation (5 in this case) the GridSearchCV looks like

```
gs_pipeline = GridSearchCV(mdl, param_grid=parameters, refit = True ,
verbose = 2, scoring=metric , cv=5 )
```

Later, the actual model is built by fitting the above with training set

```
gs_model = gs_pipeline.fit(X_train, y_train)
```

With all the mentioned preprocessing and improvements, following was the performance of improved SVC algorithm

```
[LibSVM]                    precision    recall  f1-score   support

    account related       0.61      0.65      0.63        52
appointment related       0.81      0.49      0.61        70
            billing       0.64      0.68      0.66       202
           internet       0.69      0.83      0.75       310
              phone       0.95      0.81      0.87        68
              sales       0.77      0.73      0.75       118
              video       0.90      0.75      0.82       192

        avg / total       0.75      0.74      0.74      1012
```

It can be noticed that there is a significant increase in recall and f1 scores and the accuracy is better as well.

From the gridsearch, the best parameters obtained were

C = 10 , gamma = 0.1; kernel = linear

Performance improvement for Neural network classifier

a. Increasing number of epochs

The epochs are iteration counts on the dataset. It indicates the number of scans of the entire dataset that is required to train the dataset. It's always recommended to try different values of epochs and experiment.

When no value of epoch is mentioned

```
history = nn_model.fit(NN_X_train, NN_y_train,verbose=True,
validation_data=(NN_X_test, NN_y_test))
```

```
Train on 4084 samples, validate on 1022 samples
4084/4084 [==============================] - 2s 402us/sample - loss: 1.8371 - acc: 0.3210 - val_loss:
1.7160 - val_acc: 0.3268
4084/4084 [==============================] - 0s 33us/sample - loss: 1.7151 - acc: 0.3210
Training Accuracy: 0.3210
1022/1022 [==============================] - 0s 37us/sample - loss: 1.7160 - acc: 0.3268
Testing Accuracy:  0.3268
```

When epoch is set to 1

```
history = nn_model.fit(NN_X_train, NN_y_train, epochs = 1 ,
verbose=True, validation_data=(NN_X_test, NN_y_test))
```

```
4084/4084 [==============================] - 1s 359us/sample - loss: 1.8390 - acc: 0.3198 -
val_loss: 1.6915 - val_acc: 0.3268
4084/4084 [==============================] - 0s 35us/sample - loss: 1.7026 - acc: 0.3210
Training Accuracy: 0.3210
1022/1022 [==============================] - 0s 34us/sample - loss: 1.6915 - acc: 0.3268
Testing Accuracy:  0.3268
```

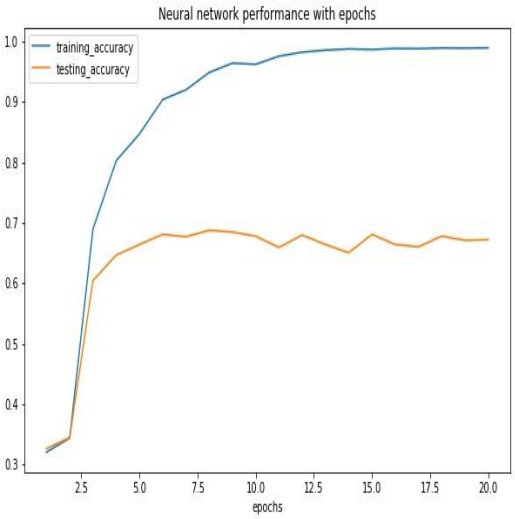
The accuracy is not changed much. The reason being when no epoch is mentioned, the default value is 1.

When epoch is set to a larger random value ‘5’

```
4084/4084 [==============================] - 0s 25us/sample - loss: 0.5512 - acc: 0.8249
Training Accuracy: 0.8249
1022/1022 [==============================] - 0s 28us/sample - loss: 0.9596 - acc: 0.6712
Testing Accuracy:  0.6712
```

The accuracy of the test data increased.

The performance of the neural network model against the number of epochs can be shown as



epochs	training_accuracy	testing_accuracy
1.0	0.321009	0.326810
2.0	0.344515	0.345401
3.0	0.689765	0.604697
4.0	0.802889	0.646771
5.0	0.847209	0.664384
6.0	0.904016	0.681018
7.0	0.920176	0.677104
8.0	0.948825	0.687867
9.0	0.964251	0.684932
10.0	0.962292	0.678082
11.0	0.975759	0.659491
12.0	0.982370	0.680039
13.0	0.985798	0.664384
14.0	0.987757	0.650685
15.0	0.986778	0.681018
16.0	0.988492	0.664384
17.0	0.988247	0.660470
18.0	0.989226	0.678082
19.0	0.988981	0.671233
20.0	0.989471	0.672211

Interestingly, the relationship between testing accuracy and number of epochs is not linear and there as some spikes observed in between. The rows that are highlighted in the right image are those corresponding spikes observed in the graph

b. Using different optimizers

Optimizers in Keras are the learning rates with which a neural network model would actually 'learn' and minimize the loss after every iteration. Keras gives options to specify either predefined (constant) optimizers or 'adaptive' optimizers.

<u>Constant learning rates:</u>

Keras provides the following constant learning rates.

a. Stochastic gradient descent

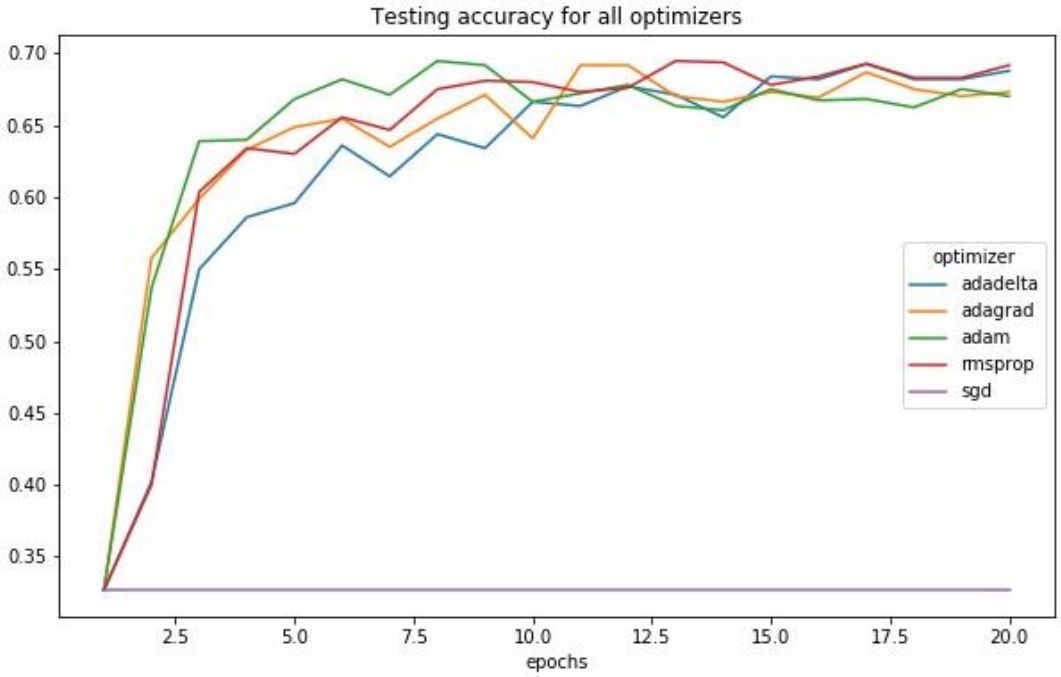
<u>Adaptive learning rates</u>

As the name suggests, these learning rate 'adapts' to the value of loss and tunes itself accordingly. Therefore, a lot of manual labor is saved. Following are types of adaptive learning rate that Keras provides

a. RMSProp optimizer
b. Adagrad optimizer
c. Adadelta optimizer.

d. Adam optimizer

Using the default parameters for these optimizers, following is testing accuracy over 20 epochs



Best performing NN optimizer and epoch

Optimizer: “rmsprop”

Epochs: 13

Testing accuracy: 0.694716215

Training accuracy: 0.88

Inference

The team has now presented the improved results and comparisons to the department head. The head is impressed with the efforts for improving the classification performance and is convinced for using Support vector classification due to its better performance. The head

suspects that a small dataset with only few thousands of sample is the reason for SVC outperforming Neural network

The svc model is then serialized into a pickle file and is ready for production.

```
svc_model = Pipeline([('vectorizer', CountVectorizer(ngram_range=(1,2))) , ('tfidf', TfidfTransformer(use_idf=True)) ,('clf', SVC(kernel='linear', C=10, gamma=0.1, verbose = True)) ])

svc_model.fit(X_train, y_train)

with open('classifier.pkl', 'wb') as fout:

      cPickle.dump(svc_model, fout)
```

Conclusion

As witnessed above, improving performance of a machine learning algorithm is an empirical process and requires a lot of experimentation. Sometimes, it's also required to analyze the sampling data and identify opportunities to clean and structure the same. Finally, in addition to a list of to-do things, intuitions and common sense can always help to raise the performance.

References

1. https://www.nltk.org/

2. https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

3. https://www.datacamp.com/community/tutorials/svm-classification-scikit-learn-python

4. https://www.datacamp.com/community/tutorials/naive-bayes-scikit-learn

5. https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

6. https://www.kaggle.com/residentmario/keras-optimizers

7. https://keras.io/optimizers/

8. https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

9. https://spacy.io/usage/linguistic-features

10. https://www.nltk.org/api/nltk.tokenize.html

11. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

12. https://scikitlearn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html