



Optimization Algorithms

for Machine Learning Models

—
Authored by: Lovekush Chaurasia, Amol Borse,
Associate Software Engineer, Noida Lead
Reviewed by: Dr. Param Jeet, Amit Sood,
Senior Data Scientist, Noida

GlobalLogic®



Contents

3	Brief
3	Introduction
4	Dataset
5	Data Cleaning
5	Convert Categorical Features to Numerical
5	Split between Test & Training Dataset
6	Training the Model
6	Checking the Model Accuracy
7	Considering Alternative Models for Classification
8	Feature Selection Using Model Importance
9	Optimizing Performance of Model Using
	Optimization algorithms
9	.K-fold Cross Validation
11	.Batch Normalization
13	.Grid Search
15	.Stochastic Gradient Descent
17	.RMSprop
19	Performance of Models After Using
	Optimization Algorithms
19	Conclusion
20	Appendix A
20	.Source Code & Datasets
20	Appendix B
20	.References

Brief

This white paper explores the optimization algorithms for machine learning models. In this use case scenario, we explore how an optimized machine learning model can be used to predict employee attrition.

Introduction

Optimization is the most crucial part of machine learning algorithms. It begins with defining loss function/cost function and ends with minimizing loss and cost using optimization algorithms. These help us maximize or minimize an error function. The internal parameters of a model play a very important role in efficiently and effectively training a model and producing accurate results. This is why we use various optimization algorithms to update and calculate appropriate and optimum values of a model's parameters. This, in turn, improves our model's learning process, as well as its output.

In this use case scenario, we'll look at machine learning in terms of employee attrition prediction. Employers generally consider attrition a loss of valuable employees and talent; however, there is more to attrition than a shrinking workforce. When employees leave an organization, they take with them much-needed skills and qualifications they developed during their tenure. There is no way for employers to know which employees will leave the company, but a well-trained machine learning model can be used to predict attrition. We will look at some of the optimization algorithms to improve the performance of the model.

Dataset

The dataset used for this analysis can be downloaded [here](#). We put it into a Pandas dataframe with the following piece of code:

```
dataset = pd.read_csv("employee_attrition.csv")
```

Taking a peek into the dataset after loading it into a Pandas dataframe. It looks something like this:

Satisfaction level	Last evaluation rating	Projects worked on	Average monthly hours	Time spend company	Work accident	Promotion last 5 years	Department	Salary
3.8	5.3	3	167	3	3.8	5.3	3	167
8	8.6	6	272	6	8	8.6	6	272
1.1	8.8	8	282	4	1.1	8.8	8	282
3.7	5.2	3	169	3	3.7	5.2	3	169
4.1	5	3	163	3	4.1	5	3	163
1	7.7	7	257	4	1	7.7	7	257
9.2	8.5	6	269	5	9.2	8.5	6	269
8.9	10	6	234	5	8.9	10	6	234
4.2	5.3	3	152	3	4.2	5.3	3	152
1.1	8.1	7	315	4	1.1	8.1	7	315
8.4	9.2	5	244	5	8.4	9.2	5	244
3.8	5.4	3	153	3	3.8	5.4	3	153
7.6	8.9	6	272	5	7.6	8.9	6	272
1.1	8.3	7	292	4	1.1	8.3	7	292
3.8	5.5	3	157	3	3.8	5.5	3	157

The dataset contains data on terminations. For each of the 10 years, it shows employees that are active and those that terminated. The intent is to see if individuals' termination can be predicted from the data provided.

Data Cleaning

The first task when analyzing any dataset is to clean the data. In our analysis, the data was already cleaned. We don't have any missing values.

```
dataset.isnull().values.ravel().sum()
```

Convert Categorical Features to Numerical

We used dummy coding to convert features to Numerical, which is a commonly used method for converting a categorical input variable into a continuous variable. Presence of a level is represented by 1, and absence is represented by 0. For every level present, one dummy variable will be created. The code for the snippet is as follows:

```
dummy_cols = ['Department', 'salary']  
X = pd.get_dummies(X, columns=dummy_cols)
```

Split Between Training and Test Dataset

The next step is to split the dataset between the training and test datasets so that we can measure the accuracy later on. We make use of the train/test split method of model selection of the Scikit-learn package to split the dataset such that the training dataset contains 70% of the observations, and the test dataset contains 30% of the observations.

```
X_train, X_test, y_train, y_test = train_test_split  
(X, y, test_size = 0.3, random_state = 0)
```

Training the Model

Once we have all our features encoded and the training dataset ready, the next step is to train a classifier model. We use the Logistic Regression method of the linear model package in Scikit-learn to train our Logistic Regression Classifier.

```
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)
```

Checking the Model Accuracy

Checking the accuracy of the model is important because it helps to see if the model is accurately predicting the true positives and the false negatives. We use the accuracy score and confidence matrix provided by the Scikit-learn library to determine the accuracy of our classifier.

In addition to this, we check Sensitivity, Specificity, F1 Score and Precision to verify that our model has good predictive power.

Following are the results of same for the Logistic Regression Model:

Model Evaluation Metric	Metric Value
Accuracy	0.78
Precision Score	0.56
Recall Score	0.31
F1 Score	0.40

Considering Alternative Models for Classification

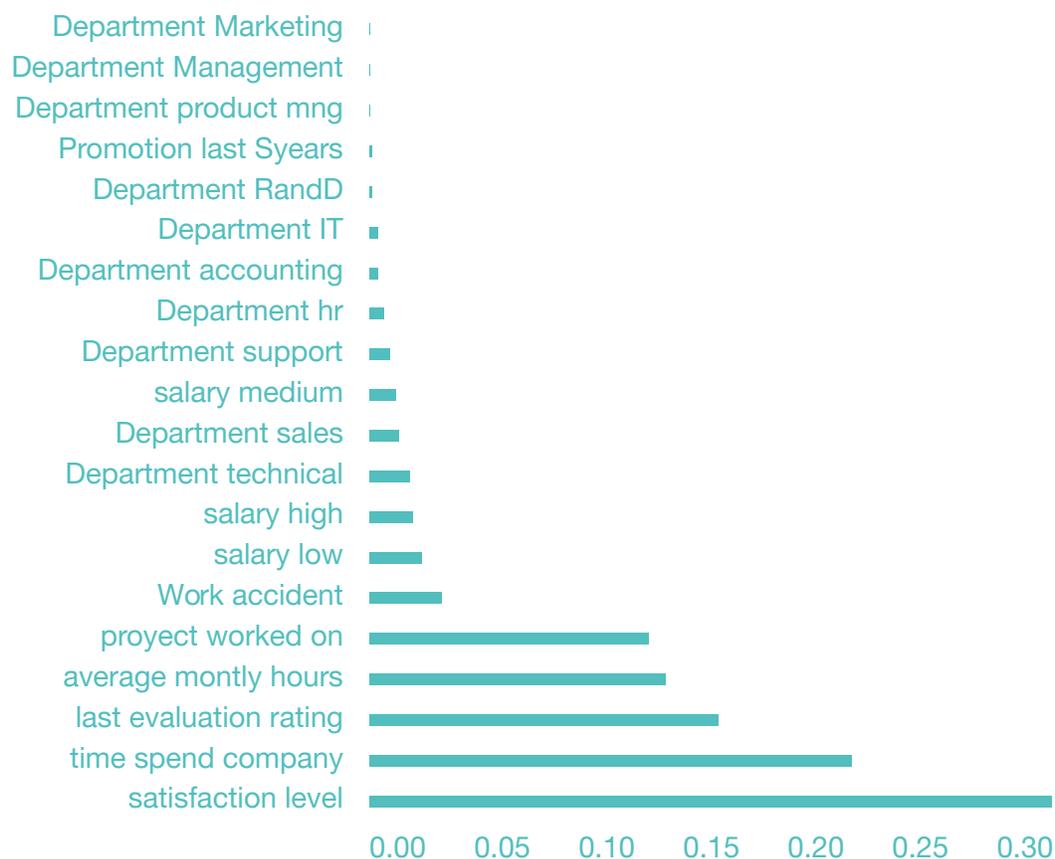
The following models were considered, along with their respective metrics:

Model	Accuracy	Precision	Recall	F1-Score	Confusion Matrix
Logistic Regression	0.78	0.56	0.31	0.40	$\begin{bmatrix} 5409 & 431 \\ 1242 & 566 \end{bmatrix}$
Random Forest Classifier	0.98	0.98	0.95	0.96	$\begin{bmatrix} 5817 & 23 \\ 84 & 1724 \end{bmatrix}$
Kernel SVM	0.94	0.87	0.88	0.88	5617
K-NN	0.95	0.86	0.93	0.89	$\begin{bmatrix} 5578 & 262 \\ 117 & 1691 \end{bmatrix}$
Naïve Bayes Classifier	0.65	0.39	0.84	0.53	$\begin{bmatrix} 3503 & 2337 \\ 289 & 1519 \end{bmatrix}$

The Random Forest classifier seems to produce the best results, so we'll optimize it using optimization algorithms.

Feature Selection Using Model Importance

We can improve the feature selection of the model by graphing the feature importance for the model. When graphing the feature importance for the Random Forest classifier, we get the following graph:



In the above bar plot, we can clearly see each of the features in the order in which they impact the employee attrition possibility. Using this information, we can drop the features that have a very low importance value as this can help to train the model faster without impacting the bias-variance trade-off.

Optimizing Model Performance Using Optimization Algorithms

K-Fold Cross-Validation

In machine learning, cross-validation is primarily used to estimate the skill of a machine learning model on unseen data. It uses a limited sample in order to estimate how the model is expected to predict in general when used to make predictions on data not used during the training of the machine learning model.

It is a popular method because it is simple to understand and because it generally results in a less biased or less optimistic estimate of the model skill than other methods, such as a simple train/test split.

Advantages of K-Fold Cross-Validation

- Cross-validation can be used to compare the performances of different predictive modeling procedures.
- Cross-validation can also be used in variable selection
- All observations are used for both training and validation, and each observation is used for validation exactly once.

Disadvantages of K-Fold Cross-Validation

- Cross-validation only yields meaningful results if the validation set and training set are drawn from the same population and only if human biases are controlled.
- Since the order of the data is important, cross-validation could be problematic for time-series models.

Batch Normalization

Batch normalization is a method used to normalize the inputs of each layer in order to fight the internal covariate shift problem, thereby improving the performance and stability of neural networks. This also makes more sophisticated deep-learning architectures.

The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be mean 0 and unit variance). This allows each layer to learn on a more stable distribution of inputs and would thus accelerate the training of the network.

We normalize the input layer by adjusting and scaling the activations, which allows each layer of a network to learn more independently of other layers.

Advantages of Batch Normalization

- **Networks train faster.** Each training iteration will actually be slower because of the extra calculations during the forward pass and the additional hyperparameters to train during back propagation. However, it should converge much more quickly, so training should be faster overall.
- **It allows higher learning rates.** Using batch normalization allows us to use much higher learning rates, which further increases the speed at which networks train.
- **It makes weights easier to initialize.** Weight initialization can be difficult, and it's even more difficult when creating deeper networks. Batch normalization seems to allow us to be much less careful about choosing our initial starting weights.

Disadvantages of Batch Normalization

- Not good for online learning
- Not good for RNN, LSTM
- Different calculation between train and test

Code Snippet

```

from keras.layers.normalization import BatchNormalization
model = Sequential()
model.add(Dense(64, input_dim=14, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(2, init='uniform'))
model.add(BatchNormalization())
model.add(Activation('softmax'))
sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=
True)
model.compile(loss='binary_crossentropy', optimizer=
sgd)
model.fit(X_train, y_train, nb_epoch=20, batch_size=16,
show_accuracy=True, validation_split=0.2, verbose = 2)

```

Implementation of Batch Normalization with Employee Attrition

```

1 X_train = sc_X.fit_transform(X_train)
2 X_test = sc_X.transform(X_test)
3
4 from keras.models import Sequential
5 from keras.layers import Dense
6 from keras.layers import BatchNormalization
7 # Initialising the ANN
8 classifier = Sequential()
9
10 # Adding the input layer and the first hidden layer
11 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 30))
12 classifier.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
13 beta_initializer='zeros', gamma_initializer='ones', moving_mean_initializer='zeros',
14 moving_variance_initializer='ones'))
15 # Adding the second hidden layer
16 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))
17
18 # Adding the output layer
19 classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
20 #getting optimizers
21
22 # Compiling the ANN
23 classifier.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = ['accuracy'])
24
25 # Fitting the ANN to the Training set
26 classifier.fit(X_train, y_train, batch_size = 32, nb_epoch = 20)
27
28 # Part 3 - Making the predictions and evaluating the model
29
30 # Predicting the Test set results
31 y_pred = classifier.predict(X_test)
32 y_pred = (y_pred > 0.5)
33
34 # Making the Confusion Matrix
35 from sklearn.metrics import confusion_matrix
36 from sklearn.metrics import f1_score, precision_score, recall_score
37 cm = confusion_matrix(y_test, y_pred)
38 print(cm)
39 precision=precision score(y_test,y_pred)

```

```

Epoch 11/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1967 - acc: 0.9372
Epoch 12/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1991 - acc: 0.9379
Epoch 13/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1983 - acc: 0.9382
Epoch 14/20
19118/19118 [=====] - 2s 108us/step - loss:
0.1945 - acc: 0.9391
Epoch 15/20
19118/19118 [=====] - 2s 105us/step - loss:
0.2006 - acc: 0.9362
Epoch 16/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1991 - acc: 0.9390
Epoch 17/20
19118/19118 [=====] - 2s 106us/step - loss:
0.1947 - acc: 0.9398
Epoch 18/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1966 - acc: 0.9382
Epoch 19/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1969 - acc: 0.9384
Epoch 20/20
19118/19118 [=====] - 2s 105us/step - loss:
0.1931 - acc: 0.9395
[[4711 180]
 [ 144 1338]]
precision:
0.8814229249611858
recall_score:
0.902634008097166
f1_score:
0.892

```

Grid-Search

Grid-searching is the process of searching the data to configure optimal parameters for a given model. There are certain parameters necessary depending on the type of model utilized. Grid-searching does not apply to only one model type. Grid-searching can be applied to calculate the best parameters to use for any given model across machine learning. It works in an iterative way. For some of the parameters associated with the model, we enter good probable values and the grid-search iterates through each of them, compares the result for each value, and then gives you the parameters best suited for your model.

Advantages of Grid Search

- Grid-search is used to find the optimal hyperparameters of a model which results in the most 'accurate' predictions.

Disadvantages of Grid Search

- Grid-search does not perform well when it comes to dimensionality; it suffers when the evaluated number of hyperparameters grows exponentially.

Code Snippet

```

from sklearn.model_selection import GridSearchCV
parameters = [{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
              {'C': [1, 10, 100, 1000], 'kernel': ['rbf'], 'gamma': [0.1, 0.2, 0.3, 0.4,
0.5, 0.6, 0.7, 0.8, 0.9]}]
grid_search = GridSearchCV(estimator = classifier,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 10,
                           n_jobs = -1)
grid_search = grid_search.fit(X_train, y_train)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_

```

Implementation of Grid Search with Employee Attrition

```

59
60 print("After OverSampling, the shape of train_X: {}".format(X_train_res.shape))
61 print("After OverSampling, the shape of train_y: {}".format(y_train_res.shape))
62
63 print("After OverSampling, counts of label '1': {}".format(sum(y_train_res==1)))
64 print("After OverSampling, counts of label '0': {}".format(sum(y_train_res==0)))
65
66 # Feature Scaling
67 from sklearn.preprocessing import StandardScaler
68 sc_X = StandardScaler()
69 X_train = sc_X.fit_transform(X_train)
70 X_test = sc_X.transform(X_test)
71
72 # Fitting Kernel SVM to the Training set
73 from sklearn.svm import SVC
74 classifier = SVC(kernel = 'rbf', random_state = 0)
75 classifier.fit(X_train, y_train)
76
77 # Predicting the Test set results
78 y_predsvm = classifier.predict(X_test)
79 Mat=sklearn.metrics.confusion_matrix(y_test,y_predsvm)
80 print(Mat)
81
82 # Grid search to find optimal parameters
83 from sklearn.model_selection import GridSearchCV
84
85 tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.01,
86 'C': [1, 10]},
87 {'kernel': ['linear'], 'gamma': [0.01, 'C': [1, 10]}]
88
89
90
91 clf = GridSearchCV(SVC(C=), tuned_parameters, n_jobs=-1, cv=5,
92                   scoring='f1_macro')
93 clf = clf.fit(X_train, y_train)
94 y_pred=clf.predict(X_test)
95
96
97 print(clf.best_params_)
98 print(clf.best_score_)

```

```

Before OverSampling, counts of label '0': 1017
After OverSampling, the shape of train_X: (67400, 7)
After OverSampling, the shape of train_y: (67400,)

After OverSampling, counts of label '1': 33740
D:\new\lib\site-packages\sklearn\preprocessing\data.py:625:
DataConversionWarning: Data with input dtype uint8, int64 were
all converted to float64 by StandardScaler.
    return self.partial_fit(X, y)
D:\new\lib\site-packages\sklearn\base.py:462:
DataConversionWarning: Data with input dtype uint8, int64 were
all converted to float64 by StandardScaler.
    return self.fit(X, **fit_params).transform(X)
main_161: DataConversionWarning: Data with input dtype uint8,
int64 were all converted to float64 by StandardScaler.
After OverSampling, counts of label '0': 33740
[[ 86 382]
 [ 24 14404]]

In [21]: from sklearn.model_selection import GridSearchCV
...:
...: tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.01,
...: 'C': [1, 10]},
...: {'kernel': ['linear'], 'gamma':
[0.01, 'C': [1, 10]}]

In [22]: clf = GridSearchCV(SVC(C=), tuned_parameters, n_jobs=-1,
cv=5,
...: scoring='f1_macro')

In [23]: clf = clf.fit(X_train, y_train)

In [24]: print(clf.best_params_)
{'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}

In [25]: print(clf.best_score_)
0.4925763171090509

In [26]:

```

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an optimization algorithm in which samples are selected randomly instead of using a whole data set for each iteration or using data in the order they appear in the training set. We adjust the weights after each iteration for our neural network.

In a typical gradient descent, the whole dataset is taken as a batch (the total number of samples from a dataset used to calculate the gradient for each iteration) which is problematic when the dataset is significantly large.. It becomes computationally expensive to perform. Stochastic gradient descent solves this problem by using a single sample to perform each iteration.

Advantages of Stochastic Gradient Descent

- It is suited for highly non-convex loss functions, such as those entailed in training deep networks for classification.
- It does the calculations faster than gradient descent and batch gradient descent.
- Stochastic gradient descent performs updates more frequently and therefore can converge faster on huge datasets.

Disadvantages of Stochastic Gradient Descent

- SGD requires a number of hyperparameters and a number of iterations.
- It is also sensitive to feature scaling.
- Error function is not well minimized.
- There is a common learning rate for all parameters.

Code Snippet

```

from keras.models import Sequential
from keras.layers import Dense
classifier = Sequential()
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu',
input_dim = 11))
classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))
classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
classifier.fit(X_train, y_train, batch_size = 10, nb_epoch = 100)

```

Implementation of Stochastic Gradient Descent with Employee Attrition

```

39
40 # Feature Scaling
41 from sklearn.preprocessing import StandardScaler
42 sc_X = StandardScaler()
43 X_train = sc_X.fit_transform(X_train)
44 X_test = sc_X.transform(X_test)
45
46 from keras.models import Sequential
47 from keras.layers import Dense
48
49 # Initialising the ANN
50 classifier = Sequential()
51
52 # Adding the input layer and the first hidden layer
53 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 11))
54
55 # Adding the second hidden layer
56 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))
57
58 # Adding the output layer
59 classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
60 from keras import optimizers
61
62 sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
63 # Compiling the ANN
64 classifier.compile(optimizer=sgd, loss = 'binary_crossentropy', metrics = ['accuracy'])
65
66 # Fitting the ANN to the Training set
67 classifier.fit(X_train, y_train, batch_size = 10, nb_epoch = 20)
68
69 # Part 3 - Making the predictions and evaluating the model
70
71 # Predicting the Test set results
72 y_pred = classifier.predict(X_test)
73 y_pred = (y_pred > 0.5)
74
75 # Making the Confusion Matrix
76 from sklearn.metrics import confusion_matrix
77 cm = confusion_matrix(y_test, y_pred)
78 print(cm)

```

```

1021: 0.1497 - acc: 0.9525
Epoch 12/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1476 - acc: 0.9521
Epoch 13/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1477 - acc: 0.9529
Epoch 14/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1461 - acc: 0.9545
Epoch 15/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1450 - acc: 0.9551
Epoch 16/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1444 - acc: 0.9544
Epoch 17/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1427 - acc: 0.9562
Epoch 18/20
19118/19118 [-----] - 2s 83us/step -
loss: 0.1424 - acc: 0.9567
Epoch 19/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1408 - acc: 0.9563
Epoch 20/20
19118/19118 [-----] - 2s 80us/step -
loss: 0.1406 - acc: 0.9568
Out[6]: <keras.callbacks.History at 0xda859e8>

In [7]: y_pred = classifier.predict(X_test)
...: y_pred = (y_pred > 0.5)

In [8]: from sklearn.metrics import confusion_matrix
...: cm = confusion_matrix(y_test, y_pred)
...: print(cm)
[[4773 118]
 [ 138 1344]]

In [8]:

```

RMSProp

The RMSprop (Root Mean Square Propagation) optimizer is similar to the gradient descent algorithm with momentum. The RMSprop optimizer restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate, and our algorithm can take larger steps in the horizontal direction and converge faster. It utilizes the magnitude of recent gradients to normalize the gradients. We always keep a moving average over the root mean squared (hence RMS) gradients, by which we divide the current gradient.

Gradients of very complex functions - such as neural networks - have a tendency to either vanish or explode as the energy is propagated through the function. The effect has a cumulative nature—the more complex the function is, the worse the problem becomes.

RMSprop is a very clever technique to deal with this problem. It normalizes the gradient itself by using a moving average of squared gradients. This balances the step size; it decreases the step for large gradients to avoid exploding, and increases the step for small gradients to avoid vanishing.

Advantages of RMSProp

- It is a very robust optimizer which has pseudo-curvature information. Additionally, it can deal with stochastic objectives very nicely, making it applicable to mini batch learning.
- It converges faster than momentum.

Disadvantages of RMSProp

- Learning rate is still manual, because the suggested value is not always appropriate for every task.

Implementation of RMSProp Descent with Employee Attrition

```

64
65 # Initialising the ANN
66 classifier = Sequential()
67
68 # Adding the input layer and the first hidden layer
69 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu', input_dim = 20))
70
71 # Adding the second hidden layer
72 classifier.add(Dense(output_dim = 6, init = 'uniform', activation = 'relu'))
73
74 # Adding the output layer
75 classifier.add(Dense(output_dim = 1, init = 'uniform', activation = 'sigmoid'))
76 #getting optimizers
77
78 # Compiling the ANN
79 classifier.compile(optimizer = 'RMSprop', loss = 'binary_crossentropy', metrics = ['accuracy'])
80
81 # Fitting the ANN to the Training set
82 classifier.fit(X_train, y_train, batch_size = 10, nb_epoch = 20)
83
84 # Part 3 - Making the predictions and evaluating the model
85
86 # Predicting the Test set results
87 y_pred = classifier.predict(X_test)
88 y_pred = (y_pred > 0.5)
89
90 # Making the Confusion Matrix
91 from sklearn.metrics import confusion_matrix
92 from sklearn.metrics import f1_score, precision_score, recall_score
93 cm = confusion_matrix(y_test, y_pred)
94 precision=precision_score(y_test,y_pred)
95 print("precision:")
96 print(precision)
97 recall=recall_score(y_test,y_pred)
98 print("recall_score:")
99 print(recall)
100 f1_score = f1_score(y_test,y_pred)
101 print("f1 score:")
102 print(f1_score)
103
19118/19118 [=====] - 2s 85us/step - loss:
0.1841 - acc: 0.9453
Epoch 11/20
19118/19118 [=====] - 2s 84us/step - loss:
0.1834 - acc: 0.9447
Epoch 12/20
19118/19118 [=====] - 2s 87us/step - loss:
0.1832 - acc: 0.9450
Epoch 13/20
19118/19118 [=====] - 2s 84us/step - loss:
0.1834 - acc: 0.9454
Epoch 14/20
19118/19118 [=====] - 2s 86us/step - loss:
0.1828 - acc: 0.9454
Epoch 15/20
19118/19118 [=====] - 2s 87us/step - loss:
0.1811 - acc: 0.9465
Epoch 16/20
19118/19118 [=====] - 2s 92us/step - loss:
0.1815 - acc: 0.9459
Epoch 17/20
19118/19118 [=====] - 2s 85us/step - loss:
0.1812 - acc: 0.9461
Epoch 18/20
19118/19118 [=====] - 2s 86us/step - loss:
0.1807 - acc: 0.9466
Epoch 19/20
19118/19118 [=====] - 2s 84us/step - loss:
0.1806 - acc: 0.9465
Epoch 20/20
19118/19118 [=====] - 2s 86us/step - loss:
0.1810 - acc: 0.9461
precision:
0.8868686868686869
recall_score:
0.888663967611336
f1_score:
0.8877654196157736
In [4]:

```

Performance of Model After Using Optimization Algorithms

Below are the optimization algorithms with their respective metrics.

Optimizers	Accuracy
K-fold	0.9740
Grid Search	0.9742
Batch Normalization	0.9395
Stochastic Gradient Descent	0.9568
RMSprop	0.9461

Conclusion

We implemented different models to predict attrition in a company, measured their accuracy, and employed the various optimization algorithms on a support vector machine to optimize its parameters. We observed that the accuracy of a model is improved by 3.4% - 94% without optimization and 97.4% with optimization using grid search. In this case, it is not a significant improvement. However, in reality we might have many more data sets where optimization improves performance significantly.

The purpose of the paper is to give an idea of various optimization techniques and how optimization helps to improve performance of any machine learning model.

Finally, we have a working model to predict which employees will leave the company and who will stay based on five input parameters with an accuracy of almost 98 percent.

Appendix A

Source Code and Datasets

Section	Data	Code
Employee Attrition	https://www.kaggle.com/alystanand/employee-attrition	https://github.com/akborse1996/Employee_Attrition

Appendix B

References

Source	Link
Building Random Forest Classifier with Python Scikit-learn	http://dataaspirant.com/2017/06/26/random-forest-classifier-python-scikit-learn/
Feature Selection	https://pdfs.semanticscholar.org/310e/a531640728702fce6c743c1dd680a23d2ef4.pdf
Hyperparameter Optimization – Siraj Raval	https://www.youtube.com/watch?v=ttE0F7fghfk
Optimizers	https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3
Batch Normalization	https://keras.io/layers/normalization/



About GlobalLogic

GlobalLogic is a leader in digital product engineering. We help our clients design and build innovative products, platforms, and digital experiences for the modern world. By integrating strategic design, complex engineering, and vertical industry expertise — we help our clients imagine what's possible and accelerate their transition into tomorrow's digital businesses.

Headquartered in Silicon Valley, GlobalLogic operates design studios and engineering centers around the world, extending our deep expertise to customers in the communications, automotive, healthcare, technology, media and entertainment, manufacturing, and semiconductor industries.

www.globallogic.com

