Restricted Boltzmann Machine

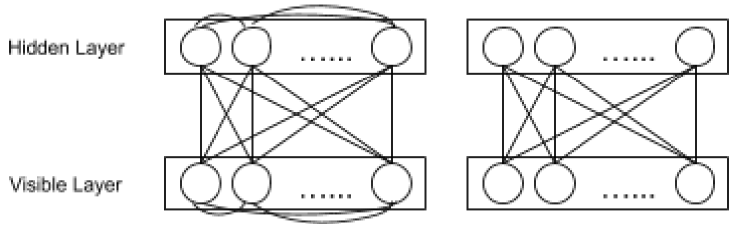
Patrick Carey, Lukasz Filipek, Taihua Li, Sriram Yarlagadda, Yuxuan Zhang

1. Introduction

Neural networks are one of the machine learning approaches, inspired by the functionalities of biological brain, used to solve complex problems. Different types of neural networks are designed for different types of data and problems. For example, the one hidden layer feedforward neural network could solve the non-linear XOR problem, and the convolutional neural network was designed to mainly learn from image data. In this paper, we will introduce Restricted Boltzmann Machine (RBM), which is an unsupervised probability based neural network. For the rest of this paper, we will first discuss the algorithm, then two tutorials of how to apply the algorithm in R and Python will be presented. At the end, two case study using Restricted Boltzmann Machine are presented.

2. Restricted Boltzmann Machine

Restricted Boltzmann Machine is a variant of Boltzmann Machines, which are stochastic recurrent neural networks invented by Geoffrey Hinton. Boltzmann Machines are designed to learn internal representations and to solve difficult combinatorial problems. However, due to its computationally expensive training process, Hinton later introduced the Restricted Boltzmann Machine, which has a two-layer bipartite neural network structure.



Boltzmann Machine (Left) vs Restricted Boltzmann Machine (Right)

As shown above, a two-layer Boltzmann Machine is a fully connected neural network where nodes in the same layer are also connected with each other. Compared to Restricted Boltzmann Machine, where the nodes in the same layer are disconnected, Boltzmann Machine needs to learn $2^{nv} + 2^{nh}$ more sets of weight, where *nv* and *nh* are the number of nodes in visible and hidden layers, respectively. Therefore, Restricted Boltzmann Machine is preferred in practice due to its reduced computational complexity.

2.1. Training a Restricted Boltzmann Machine

To train a RBM model, two phrases are involved, positive phrase and negative phrase.

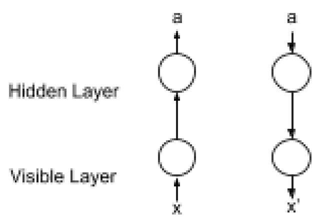
2.1.1. Positive Phrase

In the positive phrase, there are two steps involved, forward pass and backward pass, where forward pass is the same as the forward step in the feedforward network and the backward pass is a reversed forward step. The forward pass step is consisted of three components: input binary vector, weights and biases, and an activation function. Let's denote the input binary vector as *x*, weights as *w*, biases as *b*, the activation function as *σ*, and the output as *a*. Then, the forward pass can be represented as,

$$a = binarize(\sigma(xw + b)),$$

and a visual representation of the forward pass is shown on the right. In the backward pass, there are three components involved as well: activated value from the forward pass, the same sets of weights and biases, and the same activation function used in the forward pass. Let's denote a new variable *x'* as the reconstructed visible layer vector, as shown in the same graph above. Then, the backward pass can be represented as,

$$x' = binarize(\sigma(aw + b)).$$

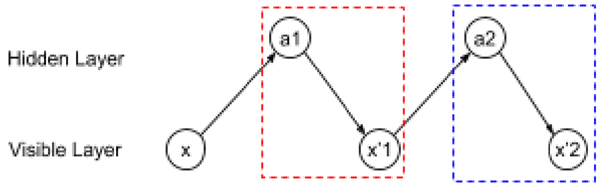


Forward Pass (left) vs Backward Pass (right)

For the activation function, the most commonly used is the *Sigmoid* or logistic function, which converts any numerical value into a value between 0 and 1. However, many other activation functions are available for different purposes. By repeating the positive phrase, forward and backward passes, Gibbs Sampling is performed to obtained different *a* and *x'* values, which will be used in the negative phrase.

2.1.2 Negative Phrase

The negative phrase in the training process is the backpropagation step of training a neural network and the learning algorithm for weights and biases is gradient learning. However, the error is computed differently for RBM since gradient ascent is used instead of gradient descent. As mentioned previously, Gibbs Sampling produces many values for *a* and *x'* at different time, *t*. With values of *a* and *x'* in the same positive phrase at time *t*, a positive gradient can be calculated. At time *t+1*, with values of *a* and *x'*, a negative gradient can be calculated. A visual representation can be shown as below,



Positive (Red) vs Negative (Blue) Gradients

With positive and negative gradients, weights and biases can be updated as,

$$w' += \varepsilon(\nabla p - \nabla n)$$

$$b_v' += \varepsilon(x'_t - x'_{t+1})$$

$$b_h' += \varepsilon(a_t - a_{t+1})$$

where ε is the learning rate for gradient learning, ∇p and ∇n are positive and negative gradients, x'_t and x'_{t+1} are reconstructed visible layers at time t and t+1, a_t and a_{t+1} are activated values at time t and t+1, and w', b_v' and b_h' are updated weights, biases for the visible and the hidden layers, respectively. After the negative phrase is completed, positive phrase should be repeated unless the training stopping criterion are met, which might include if the number of training iteration is met and if the threshold of the reconstructed error is met.

2.2. Pros and Cons of Restricted Boltzmann Machine

Restricted Boltzmann Machine in general is a simple neural network as it has a bipartite network structure and straight-forward training process. It also counters the vanishing and exploding gradient problem in neural network because since it trains in a two-layer structure, each layer is guaranteed an increase on the lower-bound of the log likelihood of the data. In addition, due to its two-layer structure, multiple RBM models can be trained and stacked together to construct a more complex network. For example, in a three layer feedforward neural network, input and hidden layers can be trained separately without the output layer using RBM. However, Restricted Boltzmann Machine is difficult and computationally expensive to train, which is discussed in the case study section of this paper.

3. Tutorial in Applying RBM Using R

In this tutorial we explore an implementation of Restricted Boltzmann Machines (RBMs) in R. R is a popular language used by a large number of data scientists, analysts, and machine learning experts and researchers around the world. This tutorial only aims to provide a quick introduction to a simple implementation of an unsupervised variant of RBMs where we only learn the weights and ignore the biases – the code could, however, be easily modified to accommodated a learning of the biases and to situations where data labels are present (supervised learning).

In the next section, there is a quick explanation of the learning process undertaken by RBMs. Following this, we look at an implementation of it in R, and then, finally, we run a face image dataset through the RBM and visualize the weight matrix that is learnt. The ultimate objective of this tutorial is to not only demonstrate a simple implementation of RBMs in R, but also provide an intuitive understanding of how RBMs learn.

Please note that the code used in this implementation has been modified from the following post: https://www.r-bloggers.com/restricted-boltzmann-machines-in-r/

3.1. Algorithm Overview

RBMs learn through an algorithm called Contrastive Divergence (CD), which was originally proposed by Geoffrey Hinton in 2002. CD makes the learning the weights and biases in RBMs and by extension other Neural Networks quick and efficient. Central to CD is the concept of Gibbs Sampling, which is a Markov Chain Monte Carlo technique used to approximate the probabilities of success (the value 1) at the different neurons in the network:

Step1: We create a blank RBM, which is a bipartite graph with a layer that corresponds to the known inputs and a second layer that corresponds to a hidden set of features. The biases (b_v and b_h) and weights (W) are initialized randomly.

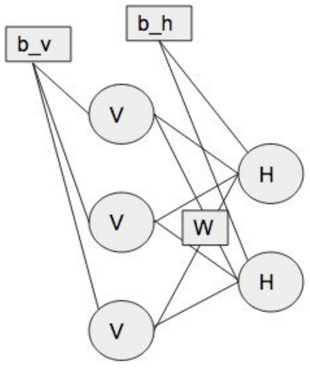


Figure: A visual representation of a simple RBM with 3 visible nodes and 2 hidden nodes

The next steps then performed for each of the training cases.

Step 2: For a given input vector V0, we calculate the probabilities at the hidden layer using a sigmoid activation, by multiplying the W matrix with the input vector and adding the biases. These probabilities are then binarized using a binomial distribution, H0. This is referred to as the forward pass.

$$H = \text{binary}(\sigma((V*W+b\_h)))$$

NOTE: In the case of a supervised variant of RBMs, we can skip this step and assign H0 a value of the label vector – the number of hidden nodes will have to be the same as that of the number of values in label vector.

Step 3: The same process is repeated in the opposite direction by using H0 as the input using the same weight matrix and biases to estimate values at the visible layer (V1).

$$V = \text{binary}(\sigma((H*W^T+b\_v)))$$

Step 4: We repeat Step 2 to calculate another estimate of the values of the hidden layer (H1). This entire process of generating H0, V1, and H1 from a known V0 is known as Gibbs Sampling.

Step 5: We now calculate the outer products of V0 and H0 (called the positive gradient, vh0) and V1 and H1 (called the negative gradient, vh1). The outer product will result in a matrix of the same shape as the weight matrix.

Step 6: The weights and biases are now updated in the following manner:

- W = W + ε*(positive gradient - negative gradient)
- b_v = b_v + ε*(V1 - V2) and b_h = b_v + ε*(H1 - H2)

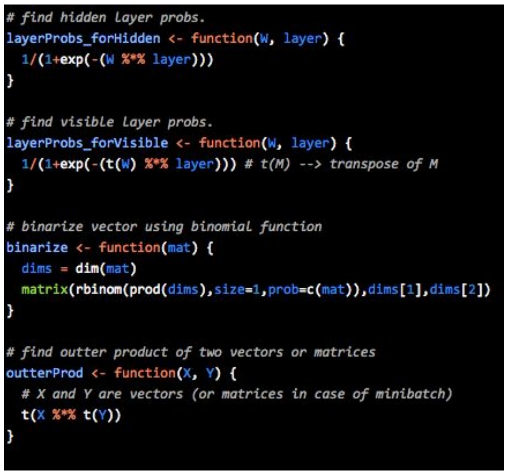
where, ε is the learning rate.

This process is repeated for all the training samples and weights and biases are progressively updated.

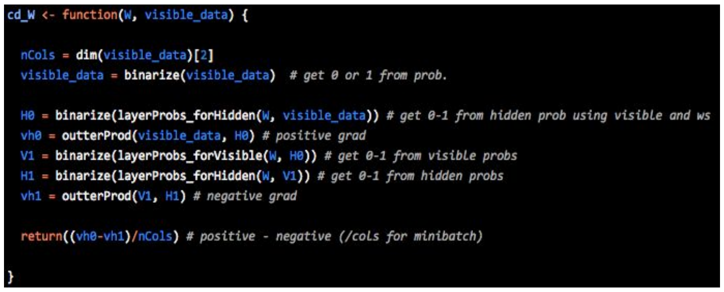
An intuitive view of this is that in the forward step, when we obtain V0 and H0, we are simply trying to get the actual representation of the values at the visible and hidden layers. And in the backward step, where we ascertain V1 and H1, we are trying to use the weight matrix and biases to find out how well they represent the data. And then by using the outer product (vh0 and vh1) we quantify this difference in an effort to change the weights and biases to bridge the gap – higher the difference, larger the updates to the weights and biases.

3.2. R Implementation

First, we create the functions needed for steps listed above. The first function shown below calculates the probabilities of the hidden layer using a sigmoid activation. The second does the same for the visible layer. The third function binarizes the probabilities by using a binomial distribution and by assuming that the probabilities obtained represent the probability of getting a1. The fourth function finds the outer product of any two given vectors or matrices.



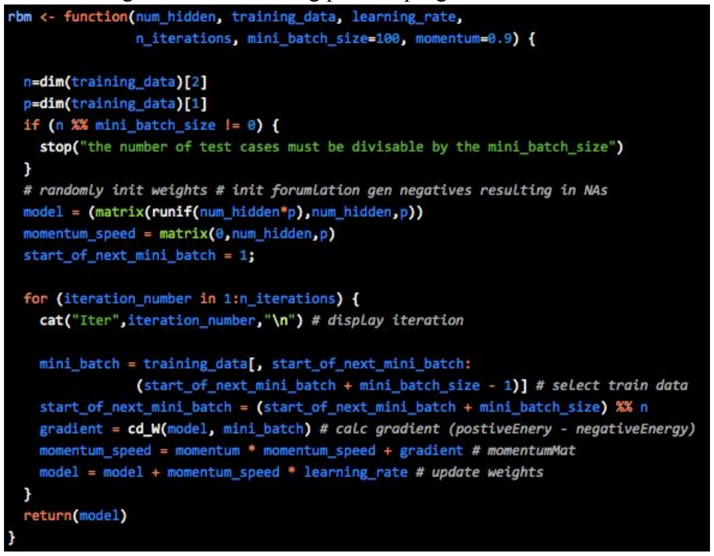
```
# find hidden layer probs.
layerProbs_forHidden <- function(W, layer) {
  1/(1+exp(-(W %*% layer)))
}

# find visible layer probs.
layerProbs_forVisible <- function(W, layer) {
  1/(1+exp(-(t(W) %*% layer))) # t(M) --> transpose of M
}

# binarize vector using binomial function
binarize <- function(mat) {
  dims = dim(mat)
  matrix(rbinom(prod(dims),size=1,prob=c(mat)),dims[1],dims[2])
}

# find outter product of two vectors or matrices
outterProd <- function(X, Y) {
  # X and Y are vectors (or matrices in case of minibatch)
  t(X %*% t(Y))
}
```



```
cd_W <- function(W, visible_data) {

  nCols = dim(visible_data)[2]
  visible_data = binarize(visible_data)  # get 0 or 1 from prob.

  H0 = binarize(layerProbs_forHidden(W, visible_data)) # get 0-1 from hidden prob using visible and ws
  vh0 = outterProd(visible_data, H0) # positive grad
  V1 = binarize(layerProbs_forVisible(W, H0)) # get 0-1 from visible probs
  H1 = binarize(layerProbs_forHidden(W, V1)) # get 0-1 from hidden probs
  vh1 = outterProd(V1, H1) # negative grad

  return((vh0-vh1)/nCols) # positive - negative (/cols for minibatch)

}
```

The next function (cd_W), first finds H0, V1, and H1, then vh0 and vh1 (the outer products), and calculates difference between the positive and negative gradients. Since we are going to be using mini batch in the actual learning loop, the gradient differences need to be divided by the number of data points in each batch (the nCols variable). Since we are only looking to learn a weight matrix, we do not update the biases in this implementation.

The function below is the main learning function that takes in the number of hidden layer nodes, data, learning rate, maximum number of iterations, and mini batch size and returns the new weight matrix (which is represented by the "model" variable in the function). The weights are constantly updated as a new batch is run through the learning functions. Momentum is also used in order to increase the learning rate as the learning process progresses.



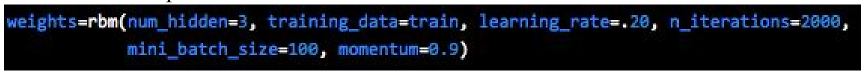
```
rbm <- function(num_hidden, training_data, learning_rate,
                n_iterations, mini_batch_size=100, momentum=0.9) {

  n=dim(training_data)[2]
  p=dim(training_data)[1]
  if (n %% mini_batch_size != 0) {
    stop("the number of test cases must be divisable by the mini_batch_size")
  }
  # randomly init weights # init forumlation gen negatives resulting in NAs
  model = (matrix(runif(num_hidden*p),num_hidden,p))
  momentum_speed = matrix(0,num_hidden,p)
  start_of_next_mini_batch = 1;

  for (iteration_number in 1:n_iterations) {
    cat("Iter",iteration_number,"\n") # display iteration

    mini_batch = training_data[, start_of_next_mini_batch:
                (start_of_next_mini_batch + mini_batch_size - 1)] # select train data
    start_of_next_mini_batch = (start_of_next_mini_batch + mini_batch_size) %% n
    gradient = cd_W(model, mini_batch) # calc gradient (postiveEnery - negativeEnergy)
    momentum_speed = momentum * momentum_speed + gradient # momentumMat
    model = model + momentum_speed * learning_rate # update weights
  }
  return(model)
}
```

The function above prints the iteration number as it is learning and since there is no tolerance threshold to stop the learning process, it keeps learning until the number of iterations are met. One thing to note is that this function needs the number of cases in the training data to be a multiple of the mini-batch size; the program will not run if that condition is not met. The function assumes that rows of the training matrix represent the features and the columns the data points. A higher learning rate would result in a quicker change in the weights, but not necessary better weight values.

3.3. Test Run

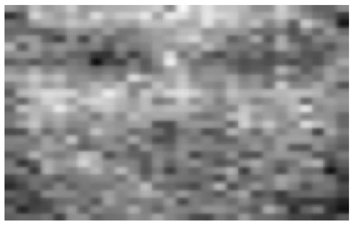
We use the learning algorithm explained above on face images to see how well it learns the weights. For this purpose, the Olivetti face image dataset (obtained from http://deeplearning.net/datasets/) has been used. The images in this dataset are in color and come in different sizes. In order to simplify the process, we resize all the images to the same size (28x28) and convert them into binary color, where 0 is white and 1 is black.

The images are unrolled into vectors of size 784 and stacked vertically. We have a total of 400 images. This results in a training data matrix of 784 x 400 shape.



```
weights=rbm(num_hidden=3, training_data=train, learning_rate=.20, n_iterations=2000,
            mini_batch_size=100, momentum=0.9)
```

The images are fed into the rbm weights learning function. We use a 3 neuron hidden layer, with a learning rate of 0.2, and mini-batch of 200 for 2000 iterations. The total run time takes about 2 minutes. Once we have the final weight matrix, we visualize the weights leading into each of the 3 hidden neurons (see below). As expected, the images appear to capture facial features.



3.4. R Tutorial Conclusion

Through this tutorial we introduced the concept of Contrastive Divergence, the learning algorithm for RBMs, and provided an intuitive explanation of how it learns. We then showed a simple implementation of this algorithm in R. We explained parts of the code in relation to the different components of the algorithm. The effectiveness of the algorithm is tested on a facial image dataset and the weight leading into the hidden neurons are visualized.

4. Tutorial in Applying RBM Using Python

In this tutorial we will be using sklearn, numpy, time, and matplotlib in a jupyter notebook. Scikit Learn library is the primary module for this tutorial as it provides the machine learning algorithms as well as the several toy datasets that are ideal for tutorial purposes. We will demonstrate not only how to build and test a classification model using Restricted Boltzmann Machines (RBM), but also how to efficiently perform hyper-parameter tuning. The hyperparameter tuning is crucial for improving accuracy and the primary cost for classification model using RBM.

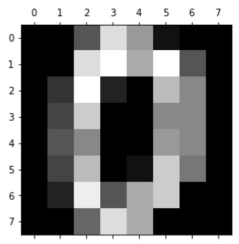
4.1. Loading Data

After loading the necessary libraries, we will load the digits dataset. The following code will load the dataset and display samples images for inspection. The digits dataset is a "Toy" dataset that comes loaded with sklearn and does not require download from any other website. This is the test dataset from the UCI Machine learning Repositories Optical Recognition of Handwritten Digits Data (http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits).

```python
import numpy as np
import matplotlib.pyplot as plt
import time
from sklearn import linear_model, svm, datasets, metrics
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.model_selection import validation_curve
```

```python
##### load digits dataset #####
digits = datasets.load_digits()
#print examples from dataset for inspection
plt.gray()
plt.matshow(digits.images[0])
plt.show()
X = digits.data
Y = digits.target

plt.figure(figsize=(10, 10))
plots = []
for i in range(10):
    for j in range(1):
        ax = plt.subplot2grid((10,1), (i,j))
        ax.imshow(digits.images[i])
plt.show()

##### alternative lfw_people dataset #####
# from sklearn.datasets import fetch_lfw_people
# lfw_people = datasets.fetch_lfw_people(min_faces_per_person=125, resize=0.3)

# for name in lfw_people.target_names:
#     print(name)

# X = lfw_people.data
# Y = lfw_people.target
# print X.shape
# plt.matshow(lfw_people.images[0])
# plt.show()

##### alternative fetch_olivetti_faces dataset #####
# from sklearn.datasets import fetch_olivetti_faces
# olivetti = fetch_olivetti_faces()
# X, Y = olivetti.data, olivetti.target
```

Optionally you can run this tutorial with the hashed out datasets: the Labeled Faces in the Wild face recognition dataset or the Olivetti Faces dataset. Descriptions for these dataset are included in the Scikit-learn documentation (http://scikit-learn.org/stable/datasets), so I won't go into further explanation here. The alternative datasets can be used to further your understanding. I did not notice a benefit to using one or the other for tutorial purposes, so we will use the digits dataset since that had the fastest run time.



One caveat I will mention about this dataset is that the original images are processed to give an 8x8 image such as the one above. This reduces the dimensionality and makes it easier to process but even the original images can be difficult to make out as you will notice from the example above.

4.2. Preprocessing and Pipelines

Next we will scale the data from 0 to 1 and add an epsilon value to prevent division by 0. The shape of the data shows there are 1,797 observations and 64 features from the 8x8 images.

```
# 0-1 scaling + epsilon value used to prevent division by zero errors.
X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001)
print X.shape
```

```
(1797, 64)
```

We will now create a pipeline for the Bernoulli RBM and Logistic Regression classifier with the above code. The alternative Support Vector Machine with a linear kernel is hashed out, which can be used later. This pipeline allows us to convert the raw pixel values with the BernoulliRBM into input in the classifier in one line of code. The parameters of the pipeline are printed for us to see for tuning purposes in the next step. While we can adjust all the parameters, we will focus on four for tutorial purposes. The hyperparameters we will focus on are:

- N_components: The number of of binary hidden units.
- Learning_rate: The learning rate for weight updates.
- N_inter: Number of iterations over the training dataset.
- Logistic_c : Inverse of regularization strength for logistic regression classifier.
- Svc_c: Penalty parameter C of the error term for the optional Support Vector Machine with a linear kernel classifier.

```
# Models we will use
logistic = linear_model.LogisticRegression()
rbm = BernoulliRBM(random_state=0, verbose=False)
classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])

#alternative SVM with linear kernel
# svc = svm.SVC(kernel='linear')
# classifier = Pipeline(steps=[('rbm', rbm), ('svc', svc)])

#print list of parameters that can be adjusted
classifier.get_params().keys()
```

```
['rbm__learning_rate',
 'logistic__C',
 'logistic__max_iter',
 'logistic__tol',
 'logistic__dual',
 'logistic__fit_intercept',
 'logistic__multi_class',
 'logistic',
 'logistic__solver',
 'rbm__n_iter',
 'rbm__n_components',
 'logistic__random_state',
 'rbm__random_state',
 'logistic__intercept_scaling',
 'logistic__verbose',
 'logistic__n_jobs',
 'rbm__verbose',
 'logistic__warm_start',
 'rbm',
 'steps',
 'logistic__class_weight',
 'logistic__penalty',
 'rbm__batch_size']
```

4.3. Tuning Hyperparameters

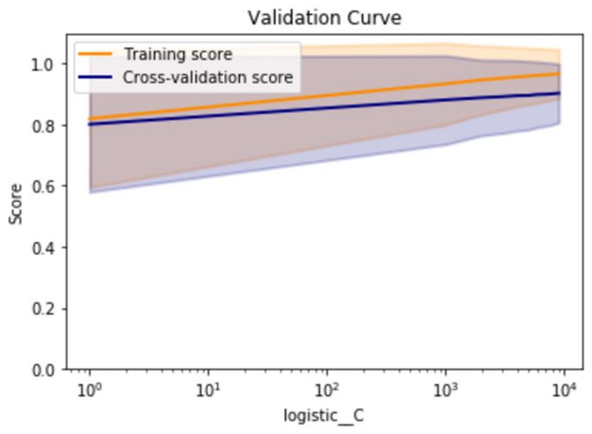
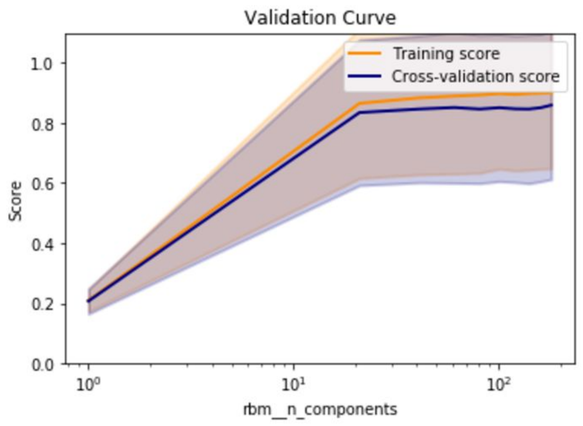
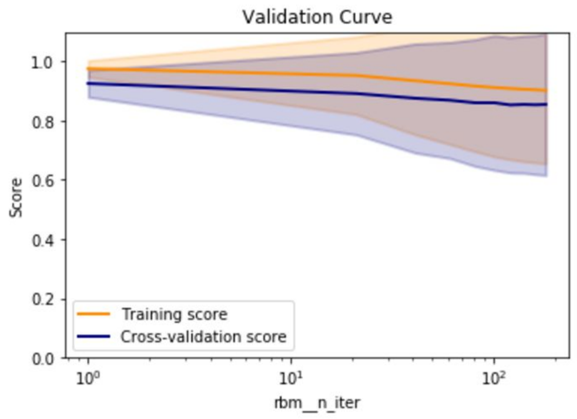
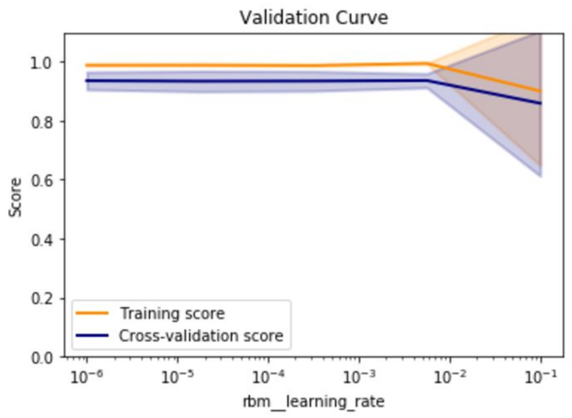
Next we will use the validation curve from Scikit-Learn to see how tuning one hyperparameter across a specified range affect the accuracy of our pipeline. It is important to note that each hyperparameter can affect the other, so we will use a Cross Validated Grid Search for the final tuning of our hyperparameters. For now, we will focus on a single hyperparameter at a time. As input you will have to place the hyperparameter name as the "p_name" variable and adjust a specified range. I have included the ranges that I used for the tutorial and noted which ranges belong to which hyperparameter. Since the dataset is small we will use the entire dataset, but I recommend using a sample for large datasets to improve runtime. The output of the code will be the validation curve using matplotlib. The score will be the accuracy, but you can adjust to any scoring method you wish. This is purely for exploration for inputs into the GridSearchCV that I will discuss later. While it always good to explore different measures, scoring measures, and graphs, you should use your time wisely and limit yourself to only what is necessary to give you an estimate of ranges for GridSearchCV.

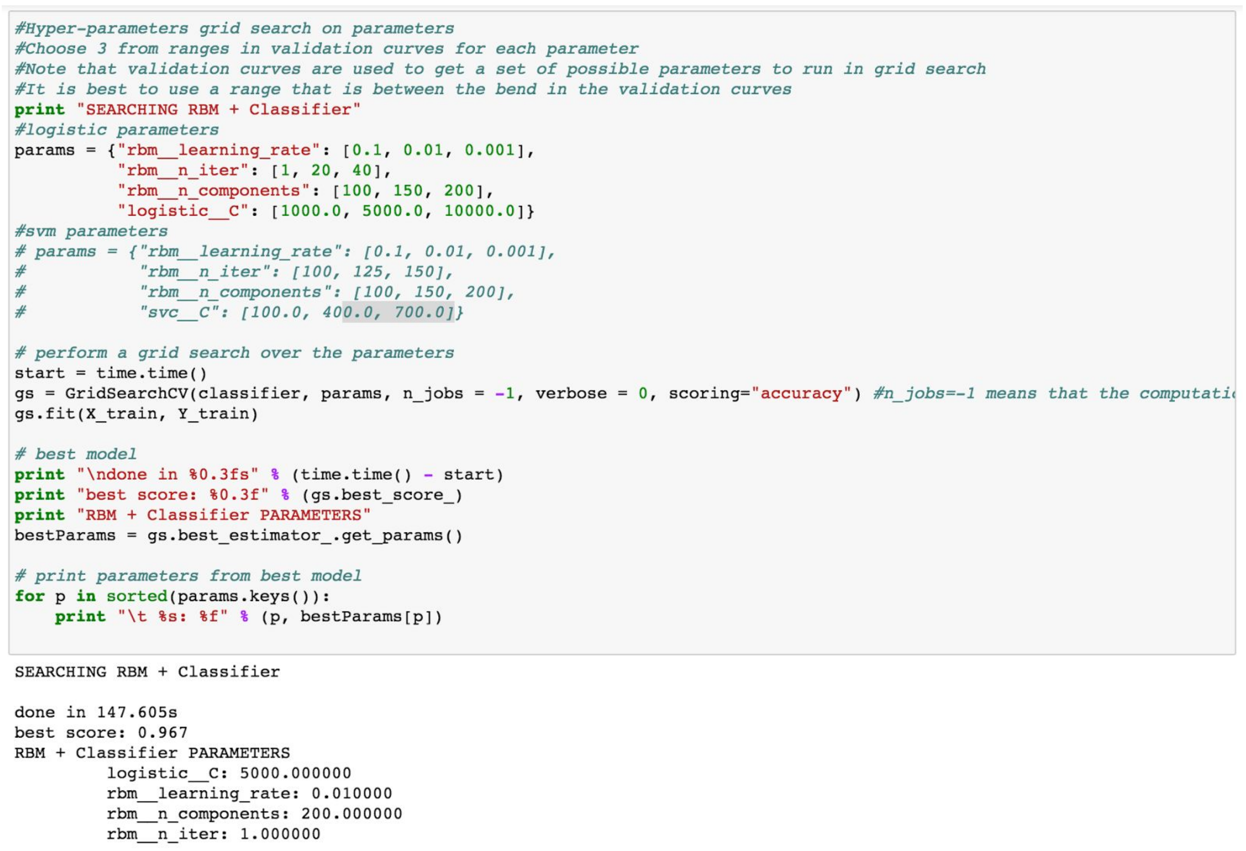
```
#Create validation curves of parameters to determine inputs for grid search cross-validation
start = time.time()
#input each parameter name
p_name = "rbm__learning_rate"#"rbm__n_components"#"rbm__n_iter" #"logistic__C"

#adjust parameter range for each variable
param_range = np.logspace(-6, -1, 5) #use this range for range for rbm__learning_rate
# param_range = range(1,202, 20) #use this range for rbm__n_iter,rbm__n_components
# param_range = range(1,10001, 1000) #use this range for range for logistic_C
# param_range = np.logspace(-10, 10, 10) #use this range for range for svm svc__C

#validation curves for analysis
train_scores, test_scores = validation_curve(
    classifier, X, Y, param_name=p_name, param_range=param_range,
    cv=10, scoring="accuracy", n_jobs=1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title("Validation Curve")
plt.xlabel(p_name)
plt.ylabel("Score")
plt.ylim(0.0, 1.1)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")

print "\ndone in %0.3fs" % (time.time() - start)
plt.show()
```

Below I have displayed the Validation Curve plots for both the RBM+Logistic Regression Pipeline and RBM+SVM Pipeline. These plots can be used to select a range for GridSearchCV. Here we will select values that centered around convergence. We will select three value one before, one during convergence, and one after if applicable. After analyzing the plots, place the list of values with the corresponding hyperparameter for GridSearchCV in the code below. We will then use these results from GridSearchCV for testing our model in the next section. We will first split our data into testing and training using Scikit-Learn's train_test_split. We will use the training data for GridSearchCV and to fit our final model. The classes for the training and testing data are displayed to confirm that all classes will be represented. For reference I have included the Validation Curve plots for the RBM + SVM pipeline at the end of this section.

```
#split into training and testing
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,test_size=0.2,random_state=0)
#check classes in training and testing
print np.unique(Y_train)
print np.unique(Y_test)
```

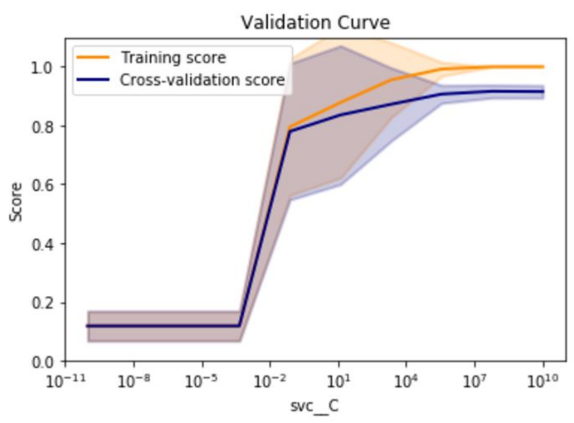
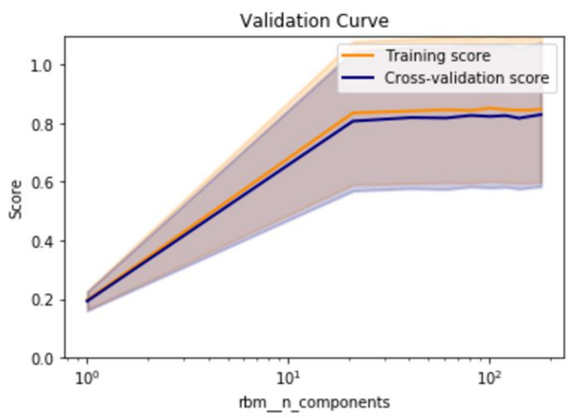
```
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
```

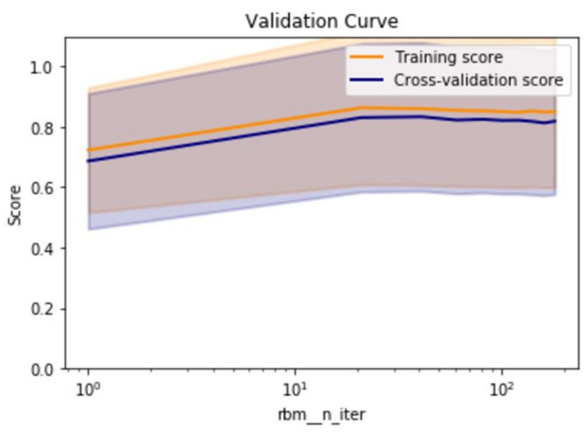
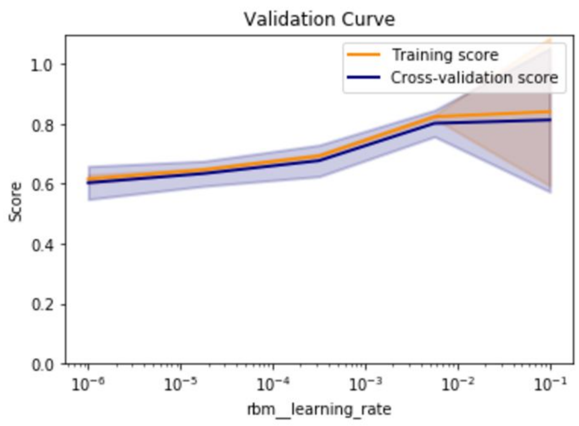
Hyperparameter Tuning with RBM + Logistic Regression Pipeline	
	
Logistic__C Values Selected for GridSearchCV: ● [1000.0, 5000.0, 10000.0]	Rbm__n_components Values Selected for GridSearchCV: ● [100, 150, 200]
	
Rbm__n_iter Values Selected for GridSearchCV: ● [1, 20, 40]	Rbm__learning_rate Values Selected for GridSearchCV: ● [0.1, 0.01, 0.001]



```
#Hyper-parameters grid search on parameters
#Choose 3 from ranges in validation curves for each parameter
#Note that validation curves are used to get a set of possible parameters to run in grid search
#It is best to use a range that is between the bend in the validation curves
print "SEARCHING RBM + Classifier"
#logistic parameters
params = {"rbm__learning_rate": [0.1, 0.01, 0.001],
          "rbm__n_iter": [1, 20, 40],
          "rbm__n_components": [100, 150, 200],
          "logistic__C": [1000.0, 5000.0, 10000.0]}
#svm parameters
# params = {"rbm__learning_rate": [0.1, 0.01, 0.001],
#            "rbm__n_iter": [100, 125, 150],
#            "rbm__n_components": [100, 150, 200],
#            "svc__C": [100.0, 400.0, 700.0]}

# perform a grid search over the parameters
start = time.time()
gs = GridSearchCV(classifier, params, n_jobs = -1, verbose = 0, scoring="accuracy") #n_jobs=-1 means that the computati
gs.fit(X_train, Y_train)

# best model
print "\ndone in %0.3fs" % (time.time() - start)
print "best score: %0.3f" % (gs.best_score_)
print "RBM + Classifier PARAMETERS"
bestParams = gs.best_estimator_.get_params()

# print parameters from best model
for p in sorted(params.keys()):
    print "\t %s: %f" % (p, bestParams[p])
```

```
SEARCHING RBM + Classifier

done in 147.605s
best score: 0.967
RBM + Classifier PARAMETERS
         logistic__C: 5000.000000
         rbm__learning_rate: 0.010000
         rbm__n_components: 200.000000
         rbm__n_iter: 1.000000
```

Hyperparameter Tuning with RBM + SVM Linear Kernel Pipeline	
	
svc__C Values Selected for GridSearchCV: ● [100.0, 400.0, 700.0]	Rbm__n_components Values Selected for GridSearchCV: ● [100, 150, 200]

	
Rbm__n_iter Values Selected for GridSearchCV: ● [100, 125, 150]	Rbm__learning_rate Values Selected for GridSearchCV: ● [0.1, 0.01, 0.001]

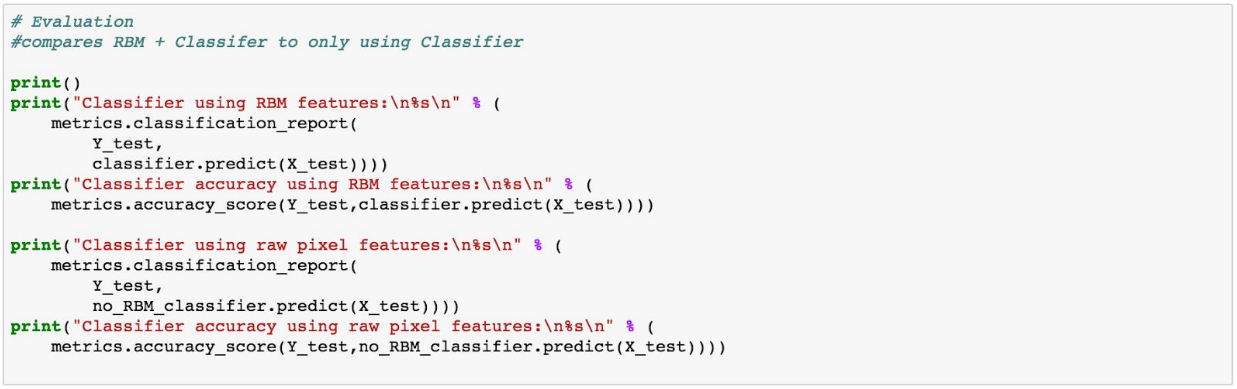
4.4. Test Results

Now we will use our pipeline with the selected hyperparameters from GridSearchCV to fit the training data. We will also fit the training data for Logistic Regression with the original pixel values for comparison.

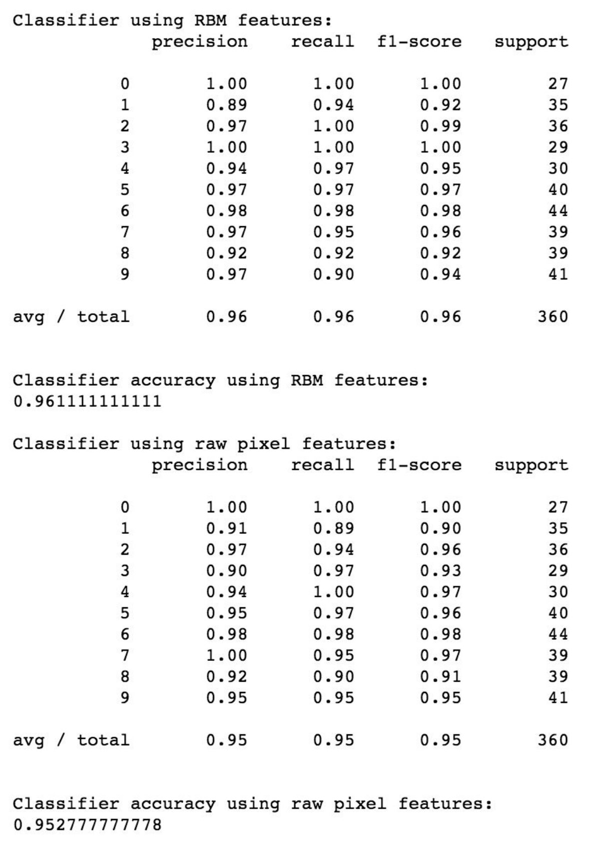
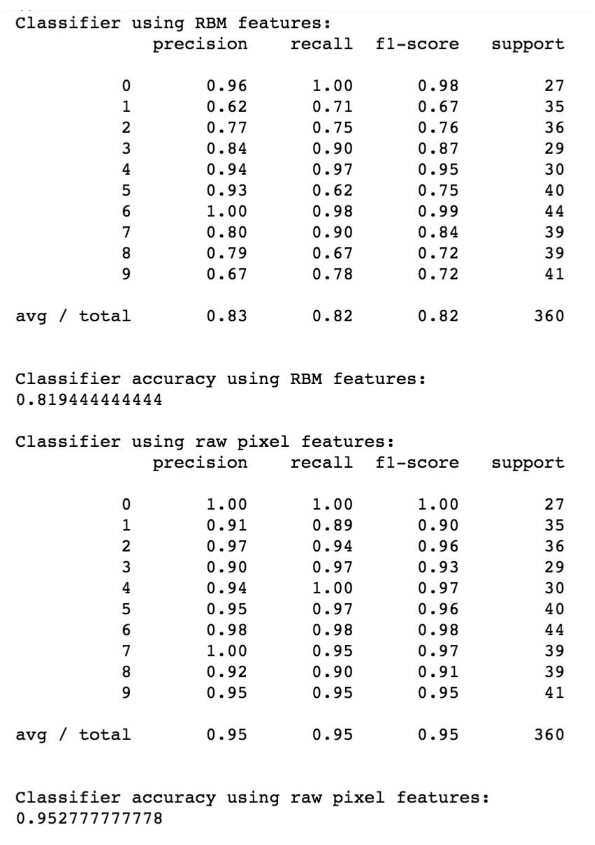
```
# Training
# set parameters from grid search
rbm.learning_rate = 0.01
rbm.n_components = 200
rbm.n_iter = 1
logistic.C = 5000.0

#parameters used from SVM grid search
# rbm.learning_rate = 0.001
# rbm.n_components = 200
# rbm.n_iter = 100
# svm.C = 100.0

# Training RBM-Classifier Pipeline
classifier.fit(X_train, Y_train)

# Training Logistic regression
no_RBM_classifier = linear_model.LogisticRegression(C=100.0)
no_RBM_classifier.fit(X_train, Y_train)

#Training SVM
# noRBM_classifier = svm.SVC(kernel='linear')
# noRBM_classifier.fit(X_train, Y_train)
```

```
LogisticRegression(C=100.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

After our data is fitted, we will run the code below to predict classes on our test data. We will use Scikit-Learn's classification and accuracy score for both models.



```python
# Evaluation
#compares RBM + Classifer to only using Classifier

print()
print("Classifier using RBM features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        classifier.predict(X_test))))
print("Classifier accuracy using RBM features:\n%s\n" % (
    metrics.accuracy_score(Y_test,classifier.predict(X_test))))

print("Classifier using raw pixel features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        no_RBM_classifier.predict(X_test))))
print("Classifier accuracy using raw pixel features:\n%s\n" % (
    metrics.accuracy_score(Y_test,no_RBM_classifier.predict(X_test))))
```

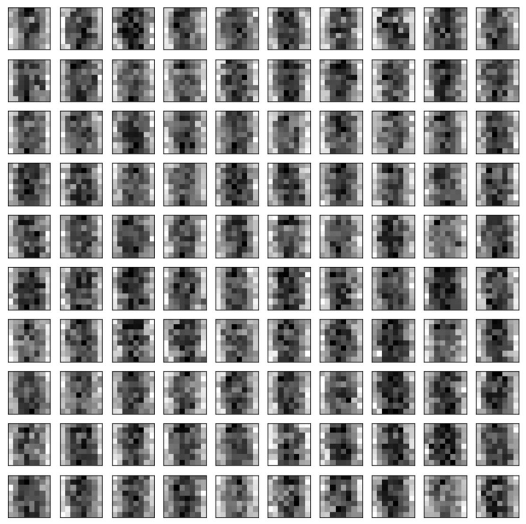
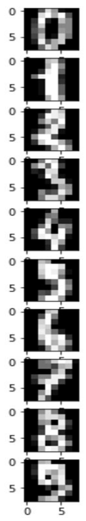
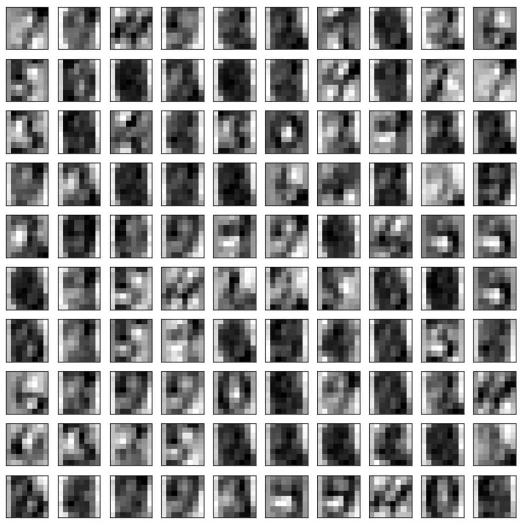
RBM + Logistic Regression	RBM + SVM with Linear Kernel
Hyperparameters: ● rbm.learning_rate = 0.01 ● rbm.n_components = 200 ● rbm.n_iter = 1 ● logistic.C = 5000.0	Hyperparameters: ● rbm.learning_rate = 0.001 ● rbm.n_components = 200 ● rbm.n_iter = 100 ● svm.C = 700.0
	

```
Classifier using RBM features:
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        27
          1       0.89      0.94      0.92        35
          2       0.97      1.00      0.99        36
          3       1.00      1.00      1.00        29
          4       0.94      0.97      0.95        30
          5       0.97      0.97      0.97        40
          6       0.98      0.98      0.98        44
          7       0.97      0.95      0.96        39
          8       0.92      0.92      0.92        39
          9       0.97      0.90      0.94        41

avg / total       0.96      0.96      0.96       360


Classifier accuracy using RBM features:
0.961111111111

Classifier using raw pixel features:
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        27
          1       0.91      0.89      0.90        35
          2       0.97      0.94      0.96        36
          3       0.90      0.97      0.93        29
          4       0.94      1.00      0.97        30
          5       0.95      0.97      0.96        40
          6       0.98      0.98      0.98        44
          7       1.00      0.95      0.97        39
          8       0.92      0.90      0.91        39
          9       0.95      0.95      0.95        41

avg / total       0.95      0.95      0.95       360


Classifier accuracy using raw pixel features:
0.952777777778
```

```
Classifier using RBM features:
             precision    recall  f1-score   support

          0       0.96      1.00      0.98        27
          1       0.62      0.71      0.67        35
          2       0.77      0.75      0.76        36
          3       0.84      0.90      0.87        29
          4       0.94      0.97      0.95        30
          5       0.93      0.62      0.75        40
          6       1.00      0.98      0.99        44
          7       0.80      0.90      0.84        39
          8       0.79      0.67      0.72        39
          9       0.67      0.78      0.72        41

avg / total       0.83      0.82      0.82       360


Classifier accuracy using RBM features:
0.819444444444

Classifier using raw pixel features:
             precision    recall  f1-score   support

          0       1.00      1.00      1.00        27
          1       0.91      0.89      0.90        35
          2       0.97      0.94      0.96        36
          3       0.90      0.97      0.93        29
          4       0.94      1.00      0.97        30
          5       0.95      0.97      0.96        40
          6       0.98      0.98      0.98        44
          7       1.00      0.95      0.97        39
          8       0.92      0.90      0.91        39
          9       0.95      0.95      0.95        41

avg / total       0.95      0.95      0.95       360


Classifier accuracy using raw pixel features:
0.952777777778
```

You will notice that we achieved an improvement in all metrics for the RBM+Logistic Regression pipeline vs Logistic Regression on the original pixel values. On the other hand, the RBM+SVM pipeline performed worse even though SVM performed well with the original pixel values. For reference, the RBM+Logistic Regression model had only an accuracy of 88% before tuning, much lower than using the just the original pixel values.

We will now view the reconstructed images from the RBM hyperparameters tuned in each pipeline. The code below will display the first 100 hidden layers from each RBM. I have included a sample of the original images for comparison. Here we can see that the hyperparameters tuned with RBM+SVM pipeline resemble much closer to the original images. This is what we would expect to see when the hyperparameters are properly tuned. It is interesting that RBM+SVM hidden layers better represent the original images, even though it performed worse for classification on our test set. Conversely, the RBM+Logistic Regression hardly resembles the digits at all and performed the best for classification. This can be due to a variety of reasons including the size of the dataset and additional tuning needed.

100 Components extracted from RBM with Logistic Regression pipeline	Original	100 Components extracted from RBM with SVM Linear Kernel pipeline
		

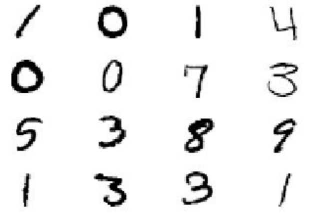
4.5. Conclusion

We have demonstrated how to analyze, efficiently tune hyperparameters, and visualize Restricted Boltzmann Machines for classification. Due to the heavy computational expense of GridSearchCV, it is important to plot valuation curves to select important ranges to cut down on computational costs. We have also demonstrated that RBM+Logistic Regression can improve accuracy and other evaluation metrics. Although RBM+SVM with a Linear Kernel had a lower accuracy, the RBM parameters were tuned much better for reconstructing the images. In this way we have also demonstrated the importance of visual analysis and metric analysis when evaluating a model. With a larger more complex dataset, it is likely that the RBM+SVM model can perform better than the RBM+Logistic model with proper tuning.

5. Case Study Using MNIST Data Set

Neural networks have become all the crazy in recent years. They have improved classification in fields such as image recognition to levels beyond human capability. Below, we explore using Restricted Boltzmann Machines (RBM's) for unsupervised features extraction of the MNIST data set for classification purposes.

5.1. Dataset



The MNIST data set is a list of 42 thousand images. The images are 28 x 28 pixels with a single channels of color (gray), totaling 284 pixels. The high dimensionality, the scale of the data set (0 – 1) and the difficulty of classifying hand written images, lend themselves very well to using neural networks.

5.2. Data Cleaning

The data came in a clean format with no missing values. Everything is flattened to a 1 dimensional vector. This is great, as it is already prepared for any algorithm. We perform a normalization to scale the data.

5.3. Data Analysis

At first glance, the dataset is relatively balanced with the following distribution:

0	1	2	3	4	5	6	7	8	9
4132	4684	4177	4351	4072	3795	4137	4401	4063	4188

We do not modify sampling to balance the classes further.

Feature extraction is the most important aspect of image classification. We will be using the RBM's to train features. The first logical step to any model building is parameter tuning. For RBM's we can use Grid Search to go through a parameter grid of the 1. Number of components in the hidden layer, 2. the number of iterations through the training dataset 3. the learning rate of every iteration. For RBM's specifically, performing a complete grid search would take hours. For example, when running a grid search, without conclusive results, on a parameter grid of 20 permutations, the code was executing for over 8 hours.
The decision is made to manually examine for convergence of 5 different rbm's with hidden layers of length 784, 392, 157, 79, 16.

Hidden Layer Components	Number of iterations	Learning Rate	Pseudo-likelihood
784	5	0.05	-12.55
392	10	0.01	-13.45
157	50	0.01	-14.10
79	50	0.01	-14.40
16	80	0.05	-14.20

We then take these different sets of features and run a proper grid search including hyper parameters for the predictive function. We choose a linear kernel SVM and logistic regression. For the purposes of this project we will explore 784 components so computation is quick. Also, the more hidden layers the potential for more intricate features to be extracted creating a more accurate model. The grids consist of C = [1,100,1000,6000] for each SVM and logistic regression.

5.4. Experimental Results

Grid search arrives at:

Logistic Regression	SVM
C =6000	C = 1000

Looking at the scores across different folds of a k-folds validation will shed light on which algorithm performs best. We perform a 5-fold test for both models and test for significance with students t-test.Our analysis concludes that Linear SVM outperforms Logistic Regression.

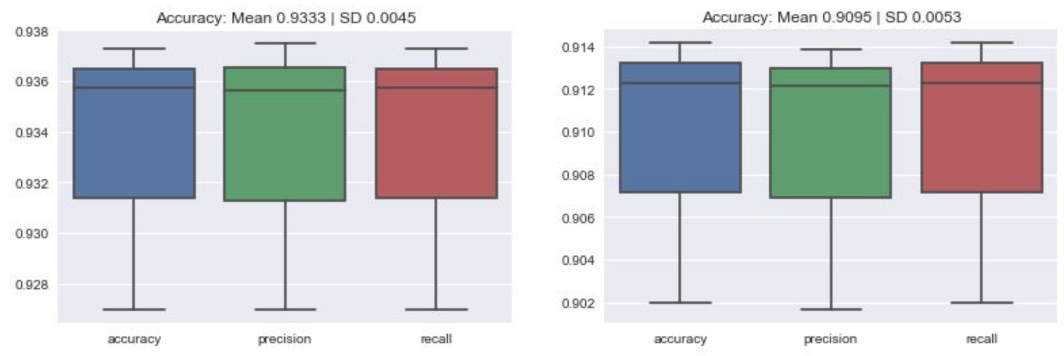
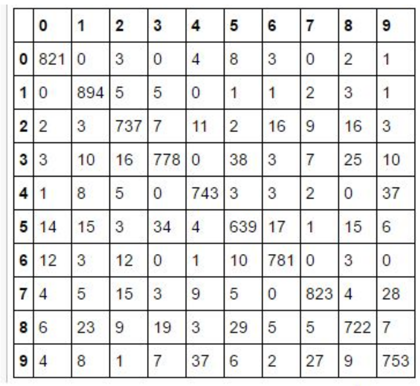


Figure: Results on SVM (left) and Logistic Regression (right)

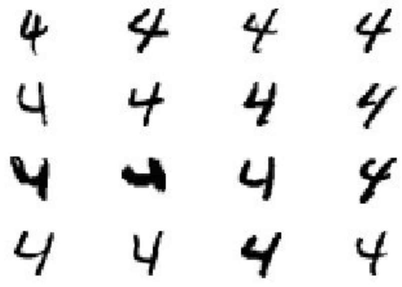
5.5. Experimental Analysis

We look at the confusion matrix of one fold. As previously mentioned, precision and recall are very good. The mistakes that are being made also subjectively make sense. For example, out of 4 prediction, 37 cases are misclassified as 9's. Many times even humans make this mistake with certain handwriting styles. We also see misclassification among 8's and 3's and 5's and 3's. An interesting misclassification is of classes 8 and 1. In this case we can see that this is occurring with very flat 8's. A human can easily see that an eight in this case is an eight; however, the neural net is not identifying the missing pixels in two centers of the digit. This speaks to potentially performing morphologies to the images to increase the training sets.

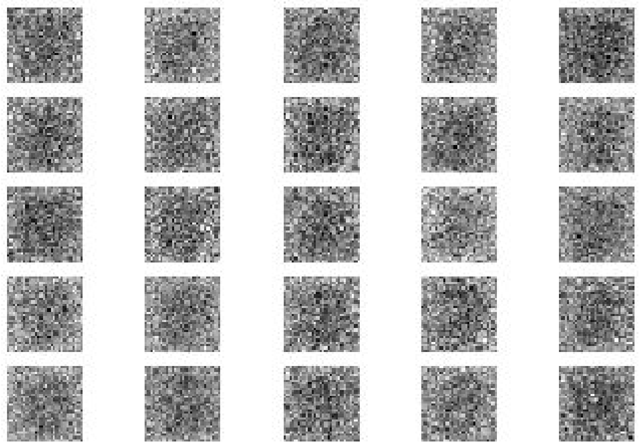


	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	821	0	3	0	4	8	3	0	2	1
1	0	894	5	5	0	1	1	2	3	1
2	2	3	737	7	11	2	16	9	16	3
3	3	10	16	778	0	38	3	7	25	10
4	1	8	5	0	743	3	3	2	0	37
5	14	15	3	34	4	639	17	1	15	6
6	12	3	12	0	1	10	781	0	3	0
7	4	5	15	3	9	5	0	823	4	28
8	6	23	9	19	3	29	5	5	722	7
9	4	8	1	7	37	6	2	27	9	753

Below we can see that 4's come in many styles. There are some that have features that are shared by many digits and therefore, it is easy to understand that some digits can get misclassified.



When observing the 25 components of the rbm, it is a bit difficult to interpret them subjectively. This may be because there are 784 components and after adding them up, we get good generation of images.



5.6. Conclusion

We can achieve good predictive results of digits using RBM's as a feature generating engine. There are certain drawbacks, the biggest one being computational power; however, the power of an algorithm creating features through generative means is extremely powerful and makes up for the computational needs. To further explore this topic, one should start with increasing the training dataset by introducing jittering, rotations and morphologies to the images to allow for better feature extraction.

6. Case Study Using 3-Dimensional Volumetric Data Set

In this second case study, we applied Bernoulli Restricted Boltzmann Machine on 3D volumetric data, with an aim to test out its ability in extracting useful latent features, reconstructing original data, and generating new data samples out of random noise.

6.1. Dataset

The volumetric data we used comes from the 3D ShapeNets dataset from Princeton University. We chose 8000 training data for chair and airplane models, respectively. Each data instance is a 32*32*32 voxelated chair/airplane model, filled with binary values indicating whether the positions are occupied with voxels or not.

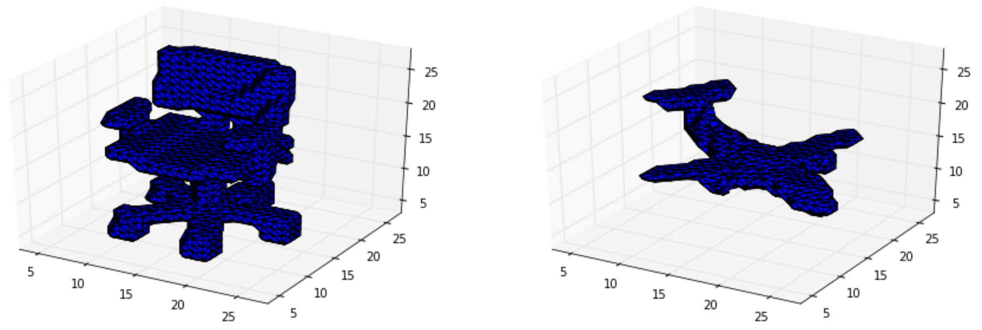


fig1. A sample training instance for chair and airplane, respectively

In other words, a training instance can be seen as a point in 32*32*32 = 32768 dimentional space, which is really sparse in this case.

6.2. Training

To train the RBM model, we used BernoulliRBM learner from Scikit-Learn. The final hyper-parameters are chosen as follows:

- n_components=1000
- learning_rate=0.02

- batch_size=200
- n_iter=200

Note that the dataset was shuffled before training to ensure enough diversity within each training iteration.

The training process ended up taking more than 16 hours on a 2.4GHz laptop. The pseudo-likelihood started from -18960.92 and gradually converged around -450, marking a significant reduction in its reconstruction error as a result of effective training.

6.3. Result

6.3.1. Activated neurons given an input

One way to check the learning effectiveness of the RBM model and interpret its learning result is to take a peek into the weight parameters associated with the hidden neuron.

To do this, we firstly transformed the original dataset (16000*32768) into its latent representation (16000*1000), where each of the latent feature is a non-linear combination of the original features/voxels. This transformation gives us a mapping between original input to the activation status of the hidden neurons, which is between 0 to 1. Once we know which hidden neuron is activated given an original instance, we can visualize the weight parameters for that particular hidden neuron as shown below:

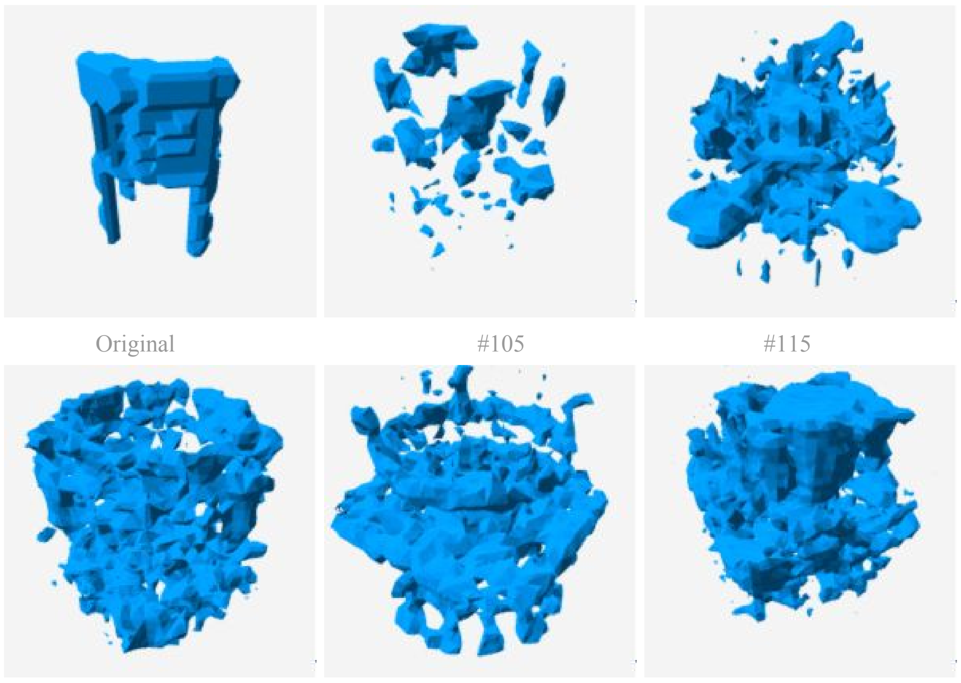


fig2. Upper left is an original chair instance, the rest are the visualization of the weight parameters corresponding to the activated hidden neurons.

As shown from fig2, five hidden neurons were activated when the chair is presented. We can visually inspect why it is the case: It seems hidden neuron #105 and #150 detects the back of the chair, #115 and #175 detects the seating platform as well as the legs.

6.3.2. 3D Model Reconstruction

Just like we can recreate a human face with principal components using PCA, we can reconstruct the 3D model using the 1000 latent features from RBM. The following shows the general reconstruction process: we start with an original instance by feeding it to the visible layer of the RBM network. Based on this input, we then calculate the possibility of the activation for each hidden neuron. Once we know which neurons are activated, we estimate the on/off statuses of the original features, that gives us a reconstructed model. This whole process is known as Gibbs sampling, we can perform it for multiple times to get a converged distribution.

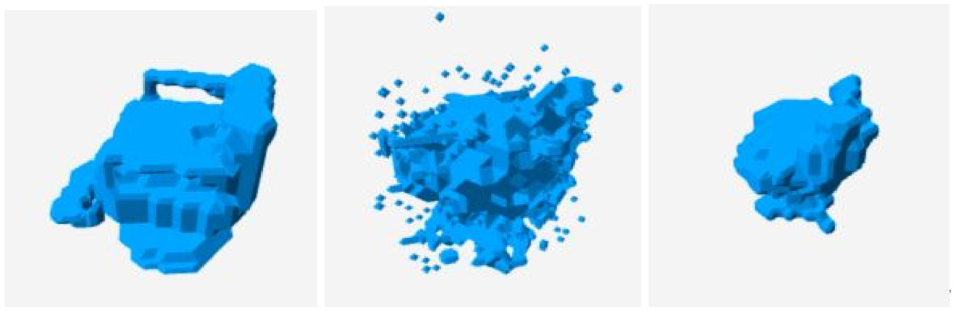


fig3. From left to right, it shows the original chair instance, reconstructed chair, and denoised chair after morphological operations.

As we can see, the reconstructed model in the middle comes with a lot of noise - isolated voxel points scattered around the main body of the model. To remove the noise and see what the reconstructed model actually looks like, we applied a 3-dimensional filter that erodes the noise and dilates the model back to its original size, the result of which is illustrated on the rightmost animation.

6.3.3. 3D Model Generation

Using the very idea of Gibbs sampling, we can perform data generation as well. In this case, instead of starting with an actual 3D model, we initiate a flattened 3D object filled with random noise points. The generation process is very similar to the reconstruction above. One thing to note is the density of the inital random noise, ideally we would want the noise density to match the density extent in the original data instances, which is around 1.5-2%.

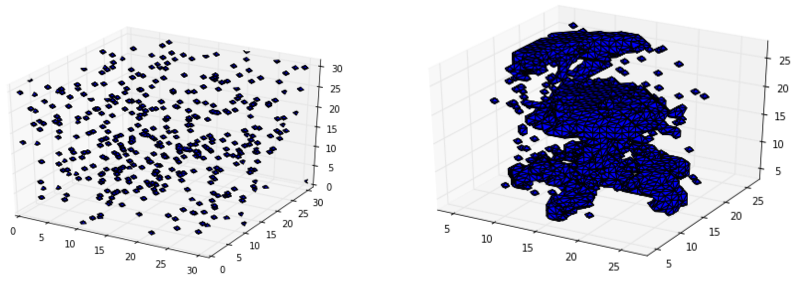


fig4. Left: inital random noise in the model; Right: generated chair after performing Gibbs sampling for multiple times.

The interesting thing is that the new instance generated doesn't necessarily match the ones in the original data, the RBM is able to capture the regularities exist in the training models and "hallucinates" about the new model. Since Gibbs sampling is a stochastic process, we usually arrive at different models each time it is performed.

7. Conclusion

In this paper, we presented Restricted Boltzmann Machine, and two tutorials of applying such model in R and Python. We also presented two case study using RBM to achieve different tasks; classification and image feature extraction. We showed that Restricted Boltzmann Machine is a powerful neural network model, especially in performing feature extraction tasks. However, the main constraint of applying such model for satisfying results is the high computational complexity of parameter tuning.

In this paper, we did not discuss several topics that can potentially boost the performance of Restricted Boltzmann Machine, and they are weight and bias initiation, learning rate setup, momentum and regularization, weight-decay, sparsity of hidden units, activation functions and types of contrastive divergence algorithms. This means there are still a lot of room for improvement for this algorithm and they should be explored and experimented in the future.

Appendix

Case study (MNIST) Code: Please see attached iPython notebook

Case Study (3D Volumetric Data) Code: https://github.com/timzhang642/3D-RBM