Introduction To Computational Intelligence

Project: Application of ANNs to Survival in the Titanic

Sunday, January 24, 2016

Erol Kazancli

Abstract

This study focuses on the application of artificial neural networks to a classification problem, specifically to classifying the passengers of Titanic as survived or not survived. The main aim is to observe the results of the application of an artificial neural network to the probably non-linear classification of multivariate data. This study shows that it is possible to obtain good enough results in such problems using artificial neural networks.

1. Introduction

Titanic is one of the most famous ships in history, in which a considerable number of people were voyaging, when against all expectations, it hit a huge iceberg and sank. In the time between the accident and the sinking of the ship, there were efforts to evacuate people with lifeboats and in this process some discrimination, the nature of which is unknown to us, was applied. As a result some people were likelier to be rescued than others.

In this project I chose Artificial Neural Networks to apply to the problem. ANNs can detect patterns in classification problems quite well and they are also robust to noise. They can be complicated to design because of the derivations and a large number of terms included. I design and train several 2-layer artificial neural networks and choose an optimum structure and model to be able to predict who were most likely to survive in the Titanic accident. I look at the characteristics like age, sex, passenger class etc., which might have had an effect on the survival. I use 5-fold cross validation to find the optimum architecture. I use misclassification error and accuracy results to compare the models. After finding the optimum model, I apply the results to the test data spared from the initial data and to the test file provided by Kaggle. I use the Python language for the programming.

2. Methods and Applications

Data Processing

The cardinality of training data set is 891. It consists of the following attributes:

Survival: (0 = No; 1 = Yes) - Integer
Pclass: Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd) - Integer
Name: Name - String
Sex: Sex – String
Age: Age - Float
Sibsp: Number of Siblings/Spouses Aboard - Integer
Parch: Number of Parents/Children Aboard - Integer
Ticket: Ticket Number - String
Fare: Passenger Fare - Float
Cabin: Cabin - String

Embarked: Port of Embarkation (C = Cherbourg; Q = Queenstown; S =Southampton) - String

I run into the general issues common in the data processing. The first of them is removing the irrelevant attributes. In my dataset, PassengerId, Name, Ticket and Cabin attributes are removed. Name and PassengerId are categorical data, which are unique for every passenger and they are very unlikely to have any effect on the survival of the passengers. Cabin and Ticket are highly varied categorical data, although there is a possibility that the Cabin information affects the survival. If I were to include these categorical attributes, I would have to add a very high number of dummy variables, which would increase the dimensionality of the data. As we know from the curse of dimensionality, a high number of dimensions necessitate more data. Besides, these attributes had a very high number of null values. Therefore it was more convenient for me to remove this type of data. The remaining data dimensions that will be used are Age, SibSp, Parch and Fare as numerical data and PClass, Sex and Embarked as categorical data. The categorical data included do not have high variability, therefore can easily be handled by dummy variables.

One other important issue with the data is the missing values for some of the attributes in the data. There are several methods to replace the null values for the preparation of the data, such as taking the mean of the values in the attribute, taking the mean of the values belonging to the same class in the attribute or removing the data containing null values. I did not have the option of removing the data since the whole data set does not contain much data. For numerical attributes I opted for the method of replacing the nanmean of the same class of the attribute for training data. For the validation and test data, I chose not to apply in-class mean because in reality we do not possess the survival information for the test data. Instead the mean of the whole attribute is taken to fill the null values. Besides, I applied these methods after separating training, validation and the test data not to bias test and validation towards the training data. For the categorical data I chose to treat null attributes as another additional category. Since the real value is unknown, it can be considered as another category that will require another dummy variable. As an alternative, filling with the mode of the attribute could also be tried. Together with all the dummy variables I have 13 dimensions plus one bias term.

Another issue with the data was normalization and standardization. We know that standardizing the input makes the training faster and decreases the risk of getting stuck at a bad local minimum. Besides, every input to an

ANN has to be numeric. The data I had was raw in that sense, having different ranges or types (numerical vs. categorical). The data used in the artificial neural networks needs to be scaled and normalized to the active range of the activation functions. The active range of the sigmoid function is $[-\sqrt{3}, \sqrt{3}]$. When the data is not within this active range, the updates to the weights are extremely small because the derivatives of a sigmoid function at the asymptotic ends are close to zero. To normalize the numerical data, I used the Gaussian normalization (z-score normalization), which is the method of subtracting the mean of the attribute from the value and dividing by the standard deviation. This method is known to be useful when there are outliers in the data. As for categorical values, since all inputs to the ANN need to be numeric they need to be converted to numerical data in an appropriate way. I used 1-of-C dummy encoding of this type of data, which is an appropriate choice for ANNs.

Lastly, for the output data, 0-1 representation is used, 0 meaning non-survival, 1 meaning survival.

Implementing the First ANN

This is my first implementation of neural networks. Because of complex derivatives used in backpropagation, I had to start small to be able to correct mistakes along the way. I started by implementing a 2-layer feedforward neural network with two hidden neurons using the equations in the books and websites. I decided mainly to use the derivations in the book Computational Intelligence, An Introduction by Adries P. Engelbrecht. My first trials were with the two-input XOR function. As the activation functions for the hidden neurons and the output neuron I used the sigmoid function, the reason of which I will explain in the following sections. Because of the complexity of the resulting derivations and the abundance of terms in the network, I initially made mistakes, which I realized thanks to the decreasing nature of the error function along the opposite side of the derivative. The mistakes were either because of one term used in the wrong place or because of the learning rate value. After I managed to converge the neural network to the correct outputs, the general structure of my neural network that will be used in my original problem of classifying the titanic passengers as survivors or non-survivors was ready.

Initializing the Architecture

The hypothesis space to be explored is two-layered feed-forward Artificial Neural Networks (one hidden layer). This hypothesis set is chosen because it is possible to approximate most classifiers with a single hidden layer, given enough number of hidden neurons and an appropriate differentiable activation function and in practice more than two layers is rarely used because the analysis gets more quite complicated.

As activation function, sigmoid function has been chosen for several reasons. One reason is that it is differentiable, which will allow backpropagation learning algorithm to be applied. Another reason is that it enables a non-linear classification because of its non-linear nature, assuming the classification is non-linear (even if it is linear, this is a subset of non-linear classification). One last reason is that sigmoid is commonly used in neural networks and a vast amount of literature can be easily found on it.

I needed a learning algorithm to be able to update the weights and biases in the direction of the correct output. For ANN's the obvious choice is the backpropagation algorithm. Thanks to the sigmoid function, the error is differentiable with respect to the input weights and hidden weights, and therefore the derivation can be easily calculated and the weights can be updated accordingly. The weights are modified in the opposite directions of these derivations by an amount of learning rate * derivation, which will decrease the error if the change is not so big as to cause a skip over the local minimum. I chose to use batch learning (by averaging the derivatives over all data to decrease the magnitude) because it was more computationally efficient in my trials. Stochastic learning turned out to be computationally costly because of weight updates for every input, which slowed my trials. I chose a learning rate of 0.15, which turned out to be good enough in my trials. The number of epochs for each fold is chosen to be 150000 because this number was more than enough for the ANN to converge to a local minimum as I observed the validation error behavior.

It goes without saying that the number of input neurons is the number of dimensions plus one (for bias). There exist 14 dimensions in total including the bias term after normalization and augmentation, therefore the network has 14 input neurons. The number of hidden neurons to be tested is chosen to be in the range of 3 to 10. The higher the number of hidden neurons, the greater is the risk of overfitting, even though overfitting can be avoided by regularization such as weight decay (which I chose not to include in my model). Moreover computational complexity increases with increasing hidden

layer size. Therefore I assumed 3-10 is a reasonable range to search for the optimum number. These models will be compared after the execution of the 5-fold cross-validation, averaging the validation errors. For the output neuron, one output neuron is used because this is a binary classification problem. Another bias is also added to connect to the output neuron along with the hidden neurons. The reason biases are added in both layers is that it is known the models with biases are more powerful allowing more flexibility.

As a rule of thumb for initializing the weights, numbers very small to zero must be used to avoid saturating the activation outputs prematurely, which will lead to zero derivatives. The range [-1/sqrt(noInputs), 1/sqrt(noInputs)] has been shown to be a good range, where "noInputs" is the number of inputs of one neuron. In addition, different random weights are chosen for each fold and hidden layer to diversify the search space for a possibility of different local minima.

Optimizing the Architecture

The training set is divided into three parts: Training, Validation and Test data. The training set is used to fit the model. On the other hand the validation set is used to stop training and compare the model performances. The test data is used to have the generalization error of the chosen model. $1/5^{th}$ of the data is spared for testing. This part of the data is not used until the end of the analysis and model/weight selection to be able to have unbiased error estimation. The other $4/5^{th}$ of the data is used for training and validation in the 5-fold cross-validation. $4/5^{th}$ of this remaining data is used for training and $1/5^{th}$ of it for validation, changing the validation and training parts for every fold. k-fold cross validation is specifically chosen to get the average validation error to be used in comparing the models. For every fold, the weights are initialized to random numbers. This 5-fold cross validation is repeated for each hidden neuron layer ranging from 3 to 10 hidden layers. For each fold the smallest validation error is picked, which would normally be the point to stop the training to avoid overfitting. But in this analysis only the minimum validation error matters, not the model weights (The model will be trained again after the optimum number of hidden neurons has been found). At the end of the analysis the hidden layer that gave the smallest average error is chosen. The result of the 5-fold cross validation analysis for each hidden layer is as follows:

	Average Validation Error
3 Hidden Neurons	0.25454545
4 Hidden Neurons	0.25874126
5 Hidden Neurons	0.23496503
6 Hidden Neurons	0.23076923
7 Hidden Neurons	0.22657343
8 Hidden Neurons	**0.22097902**
9 Hidden Neurons	0.23496503
10 Hidden Neurons	0.22377622

Table 1 - Validation Error Comparison Across Varying Hidden Layer Size

Choosing the Model

In my cross-validation tests the optimum architecture turned out to be the ANN with 8 hidden neurons and batch learning with a learning rate of 0.15. Using this architecture I trained the model to find the model weights. For the stopping criterion to be used, $1/5^{th}$ of the data is spared for the validation. The weights are initialized to random values and training is carried out until a minimum validation error is reached to stop the training. This process is repeated with different initial weights to search for different local minima until we see the best minimum validation error. The initial weights are stored for the best minimum validation error and the training is done one last time until this minimum validation error is reached. Here I used a trick by first finding the minimum validation error by training with 150000 epochs and then starting the training anew and stopping when this minimum validation error is reached. Otherwise it was quite difficult to compare the validation errors in subsequent epochs because the validation error fluctuates a lot. The weights produced by the analysis can be seen in the file ApplyModelToTest.py and ApplyModelToTestFile.py.

3. Test Results

After the application of 5-fold analysis, an optimum layer size of 8 hidden neurons was found. Using this layer size the optimum model was obtained and the model parameters are finally found. This model is then applied to the Test portion previously spared of the data and an unbiased estimate of **0.19** has been obtained, which means **%81** accuracy. The found

model is also applied to the Test file provided by the Kaggle, for which we do not have the survival information. The result obtained by the model is stored testresults.csv file. When this file was uploaded in the Kaggle website, an accuracy rate of **%73.2** was obtained for half of the data which is the only revealed part. The accuracy for the other half will be revealed at the end of the competition.

4. Possible Improvements

A single simulation in ANN's generally is not enough to produce conclusive results. Even though I have done a few simulations, more simulations with random initial weights could be done to find the right architecture and the right model given more time. Performance results could be averaged over all simulations in this case producing much better results.

The search for best initial weights could also be improved by the use of genetic algorithms. Momentum factor could also be used to converge faster and also possibly to escape bad local minima. Weight decay with a higher number of hidden neurons could also be applied to find better models.

One other issue that could be improved is the processing of the null values. Filling the null values with the in-class mean may not have been a good choice, biasing the data. Moreover, discarding some attributes because of high dimensionality or the complexity of the data may have meant a loss of important data. A good method could be found to extract information from this data as well. PCA could also be applied even though ANNs intrinsically apply PCA at the cost of more hidden neurons. By applying PCA, the hidden layer size could be reduced.

5. Conclusions

The analysis and application of the found model produced interesting results. The accuracy of the classification is not perfect but can be considered good enough. After all, the intention of CI techniques is to find approximate or good-enough results. Of course, the results could be improved with much better tuning of the model parameters and it would be interesting to see the results getting better.

On the other hand, this implementation of neural networks helped me to understand the inner workings of an ANN. There are many parameters to tune in an artificial neural network and sometimes even a slight change may affect the result. The number of hidden neurons, the initial weights, the learning rate, the processing of data, the activation functions and many more affect the process or the outcome, details of which I explained in the sections above. I had the opportunity to see the effects of these parameters. It was interesting for me to see that the backpropagation learning algorithm really works in the sense that the error decreases as the weights are updated using the derivative rules. For me, it was an illustration of how CI techniques are an alternative to conventional techniques to find a solution to real-life problems.

References

Andries P. Engelbrecht (2007), *Computational Intelligence An Introduction*. University of Pretoria South Africa

Ethem Alpaydın(2010), *Introduction to Machine Learning*. The MIT Press, Cambridge, Massachusettes

Christopher M. Bishop(2006), *Pattern Recognition and Machine Learning*.

David Kriesel(2005), *A Brief Introduction To Neural Networks*. Retrieved from http://www.dkriesel.com/en/science/neural_networks

Tom M. Mitchell(1997), *Machine Learning*. McGraw Hill

Martin T. Hagan, *Neural Network Design*. Oklahoma State University. Retrieved from http://hagan.okstate.edu/NNDesign.pdf

Professor Anita Wasilewska, *Lecture Notes on Neural Networks*. Retrieved from http://www3.cs.stonybrook.edu/~cse634/ch6NN.pdf

Implemeting a Neural Network from Scratch in Python. Retrieved from http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/