Forecast Use Of A City Bikeshare System

Darshan Washimkar

December 15, 2014

Contents

1 Introduction **1**
- 1.1 Motivation For The project . . . 1
- 1.2 Problem Description . . . 2
- 1.3 Data Description . . . 2

2 Implementation **3**
- 2.1 Python Code To Read The Data . . . 3
- 2.2 Methods Used . . . 3
  - 2.2.1 Regularized Linear Least Squares . . . 3
  - 2.2.2 Nonlinear Regression with Neural Networks . . . 4
- 2.3 Python Implementation of Models . . . 5
  - 2.3.1 Python Implementation of Linear Model . . . 5
  - 2.3.2 Python Implementation of Neural Networks . . . 5

3 Experiment **8**
- 3.1 Direct Count Prediction . . . 8
- 3.2 Indirect Count Prediction . . . 8

4 Results **12**

Abstract

This document is a project report submitted as a part of CS545 class. This report gives overview about project idea and explain the bike sharing systems used all over the world. It also gives introduction to the methods used to solve given problem of predicting bike share demand. I also presented the python code used for reading data. In the last section, result are explained and analysis is presented.

1 Introduction

This Section illustrate a problem statement for the project. This project is one of problem available for competitions on `https://www.kaggle.com/`. Kaggle is a community of data scientists who competes with each other to solve complex data science problems.

1.1 Motivation For The project

I find regression analysis as a most interesting concept in machine learning. Regression analysis is a process for estimating the relationships between variables. There are many techniques that can help in modeling and analyzing several variables. More specifically, regression analysis helps one understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are kept constant.

There are different liner and non-liner regression modeling technique that we learned in the class. Regression with Neural Networks is one of the most interesting method to me. In order to investigate and learn

more about this method, I chose 'Bike Sharing Demand' problem listed on `https://www.kaggle.com/`. It is an interesting regression problem where bike rental demand in the Capital Bikeshare program need to be predicted based on historical usage and weather data. It is an interesting problem also because it has twelve input parameters and many parameters have different classes in itself. Project details and data description are given in following subsections.

1.2 Problem Description

A bicycle sharing system is a service in which bicycles are made available for shared use to individuals on a very short term basis. Bike share schemes allow people to borrow a bike from point "A" and return it at point "B". The process of obtaining membership, rental, and bike return is automated via connected network. In April 2013 there were around 535 programs around the world, made of an estimated fleet of 517,000 bicycles.

The data generated by these systems makes them attractive for researchers because the duration of travel, departure location, arrival location, and time elapsed is explicitly recorded. Bike sharing systems therefore function as a sensor network, which can be used for studying mobility in a city. Objective of this project is to forecast bike rental demand in the Capital Bikeshare program in Washington, D.C. by using historical usage patterns and weather data. I need to build the best model to predict the total count of bikes rented during each hour covered by the test set, using only information available prior to the rental period.

1.3 Data Description

Data for the project is provided by Hadi Fanaee Tork using data from Capital Bikeshare. (`http://www.capitalbikeshare.com/system-data`) It contains hourly rental data spanning two years. The training set is comprised of the first 19 days of each month, while the test set is the 20th to the end of the month.

Data available for this project contains following data fields,

- **Datetime** - hourly date + timestamp
- **Season** - 1 = spring, 2 = summer, 3 = fall, 4 = winter
- **Holiday** - whether the day is considered a holiday
- **Workingday** - whether the day is neither a weekend nor holiday
- **Weather** -
  - 1 = Clear, Few clouds, Partly cloudy, Partly cloudy
  - 2 = Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3 = Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4 = Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- **Temp** - temperature in Celsius
- **Atemp** - "feels like" temperature in Celsius
- **Humidity** - relative humidity
- **Windspeed** - wind speed
- **Casual** - number of non-registered user rentals initiated
- **Registered** - number of registered user rentals initiated
- **Count** - number of total rentals

Absence of 'Casual' and 'Registered' data field in the test data set makes this problem more interesting.

2 Implementation

This section gives an overview of the methods used to solve bike sharing demand problem. It also gives the python code for reading the data.

2.1 Python Code To Read The Data

I have used numpy library to read training and testing data. Below is a code to that reads a training data and create input and target matrices to training the model.

```
D = np.loadtxt('train.csv',delimiter=",", converters={0:strpdate2num('%Y-%m-%d %H:%M:%S')},
    skiprows=1)
T = D[:,-1]
T = T.reshape(T.shape[0],1)
DD = D[:,1:9]
Date_all = num2date(D[:,0])
weekday = []
Time_Of_Day = []
for date in Date_all:
    weekday.append(date.weekday())
    Time_Of_Day.append(date.hour)
week_day = np.array(weekday)
time_of_day = np.array(Time_Of_Day)
Data = np.hstack((DD, week_day.reshape(week_day.shape[0],1),
    time_of_day.reshape(time_of_day.shape[0],1)))
```

Following is a code to read the testing data.

```
TT = np.loadtxt('test.csv',delimiter=",", converters={0:strpdate2num('%Y-%m-%d %H:%M:%S')},
    skiprows=1)
TTT = TT[:,1:10]
Date_all = num2date(TT[:,0])
weekday = []
Time_Of_Day = []
for date in Date_all:
    weekday.append(date.weekday())
    Time_Of_Day.append(date.hour)
week_day = np.array(weekday)
time_of_day = np.array(Time_Of_Day)
Data_Testing = np.hstack((TTT, week_day.reshape(week_day.shape[0],1),
    time_of_day.reshape(time_of_day.shape[0],1)))
```

2.2 Methods Used

This subsection explains the linear and non-linear model that is used to solve bike sharing demand problem.

2.2.1 Regularized Linear Least Squares

linear least squares is an approach fitting a mathematical or statistical model to data in cases where the idealized value provided by the model for any data point is expressed linearly. The most important application of linear least squares is in data fitting.[]

Deriving the expression for the weights that minimized the sum of squared errors of a linear model: y being an affine (linear + constant) function of x,

$$y(x; w) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D = w^T x$$

having parameters $w = (w_0, w_1, w_2, \ldots, w_D)$, we derived the solution to

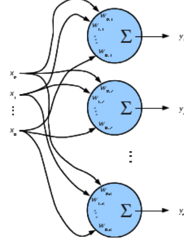

Figure 1: Simple Neural Network

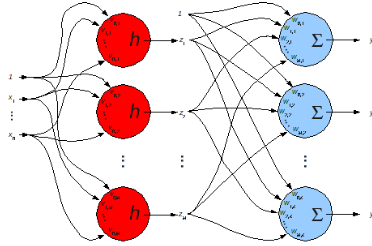

Figure 2: Structure of a Two Layer Neural Network

$$w_{\text{best}} = argmin_w \sum_{n=1}^{N} (t_n - x_n^T w)^2$$

If we add λ to this error measure, that ends up minimizing the magnitude of the weights. In matrix form this will look like,

$$(T - Xw)^T (T - Xw) + \lambda w^T w$$

By taking the derivative and equating it to zero, we will end up with following equation.

$$w = (X^T X + \lambda I)^{-1} X^T T$$

2.2.2 Nonlinear Regression with Neural Networks

Artificial neural networks (ANNs) are computational models inspired by an animal's central nervous systems (in particular the brain), and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Figure 1: Simple Neural Network shows a simple example of neural network where every input features is decisive of each target feature depending of model weights.

In the Figure 2: Structure of a Two Layer Neural Network, two layers are called the hidden and output layer. h is the activation function for the units in the hidden layer. We will be doing gradient descent in the squared error, so want an h whose derivative doesn't grow out of control as v grows, and whose derivative is easy to calculate.

Figure 3: Training Neural Network with Back-Propagation and following expressions illustrate the overall process of training neural network with back-propagation. Weights of each hidden units are updated by calculating error at previous layer.

$$\begin{aligned}
Z &= h(\tilde{X}V) \\
Y &= \tilde{Z}W \\
V &\leftarrow V + \rho_h \frac{1}{N} \frac{1}{K} \tilde{X}^T \left((T - Y)\hat{W}^T \cdot (1 - Z^2) \right) \\
W &\leftarrow W + \rho_o \frac{1}{N} \frac{1}{K} \tilde{Z}^T (T - Y)
\end{aligned}$$

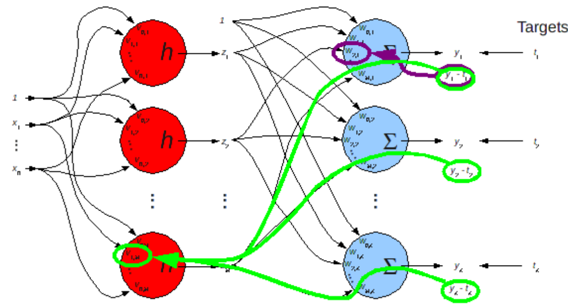

Figure 3: Training Neural Network with Back-Propagation

2.3 Python Implementation of Models

2.3.1 Python Implementation of Linear Model

```
def makeLLS(Xtrain,Ttrain,lamb):
   (standardize, unStandardize) = makeStandardize(Xtrain) ##Standardize X
   XtrainS = standardize(Xtrain)
   XtrainS1 = np.hstack((np.ones((XtrainS.shape[0],1)), XtrainS))
   lambdaI = lamb * np.eye(XtrainS1.shape[1]) ##Adding column of 1's to X
   lambdaI[0,0] = 0
   w = np.linalg.lstsq( np.dot(XtrainS1.T, XtrainS1) + lambdaI, np.dot(XtrainS1.T, Ttrain))[0]
       ##Calculate weights
   return {'standardize':standardize,'w':w}

def makeStandardize(X):
   means = X.mean(axis=0)
   stds = X.std(axis=0)

      def standardize(origX):
      return (origX - means) / stds

      def unStandardize(stdX):
      return stds * stdX + means

   return (standardize, unStandardize)

def useLLS(model,X):
   X1 = np.hstack((np.ones((X.shape[0],1)), model['standardize'](X)))
   ## Use the function standardize in model to standardize X
   ## Use weights in model to calculate predictions
   return np.dot( X1, model['w'] ) ## Returning predictions
```

2.3.2 Python Implementation of Neural Networks

```
class NeuralNetwork:
    def __init__(self,ni,nhs,no):
        try:
            nihs = [ni] + list(nhs)
        except:
            nihs = [ni] + [nhs]
            nhs = [nhs]
```

```
        self.Vs = [np.random.uniform(-0.1,0.1,size=(1+nihs[i],nihs[i+1])) for i in
            range(len(nihs)-1)]
        self.W = np.random.uniform(-0.1,0.1,size=(1+nhs[-1],no))
        # print [v.shape for v in self.Vs], self.W.shape
        self.ni,self.nhs,self.no = ni,nhs,no
        self.Xmeans = None
        self.Xstds = None
        self.Tmeans = None
        self.Tstds = None
        self.iteration = mp.Value('i',0)
        self.trained = mp.Value('b',False)
        self.reason = None
        self.errorTrace = None

    def getSize(self):
        return (self.ni,self.nhs,self.no)

    def getErrorTrace(self):
        return self.errorTrace

    def getNumberOfIterations(self):
        return self.numberOfIterations

    def train(self,X,T,
              nIterations=100,weightPrecision=0,errorPrecision=0,verbose=False):
        if self.Xmeans is None:
            self.Xmeans = X.mean(axis=0)
            self.Xstds = X.std(axis=0)
        X = self._standardizeX(X)

        if T.ndim == 1:
            T = T.reshape((-1,1))

        if self.Tmeans is None:
            self.Tmeans = T.mean(axis=0)
            self.Tstds = T.std(axis=0)
        T = self._standardizeT(T)

        # Local functions used by gradientDescent.scg()

        def objectiveF(w):
            self._unpack(w)
            Y,_ = self._forward_pass(X)
            return 0.5 * np.mean((Y - T)**2)

        def gradF(w):
            self._unpack(w)
            Y,Z = self._forward_pass(X)
            delta = (Y - T) / (X.shape[0] * T.shape[1])
            dVs,dW = self._backward_pass(delta,Z)
            return self._pack(dVs,dW)

        scgresult = scg.scg(self._pack(self.Vs,self.W), objectiveF, gradF,
                            xPrecision = weightPrecision,
                            fPrecision = errorPrecision,
                            nIterations = nIterations,
                            iterationVariable = self.iteration,
                            ftracep=True,
                            verbose=verbose)
```

```
        self._unpack(scgresult['x'])
        self.reason = scgresult['reason']
        self.errorTrace = scgresult['ftrace']
        self.numberOfIterations = len(self.errorTrace) - 1
        self.trained.value = True
        return self

    def use(self,X,allOutputs=False):
        Xst = self._standardizeX(X)
        Y,Z = self._forward_pass(Xst)
        Y = self._unstandardizeT(Y)
        return (Y,Z[1:]) if allOutputs else Y

    def draw(self,inputNames = None, outputNames = None):
        ml.draw(self.Vs, self.W, inputNames, outputNames)

    def __repr__(self):
        str = 'NeuralNetwork({}, {}, {})'.format(self.ni,self.nhs,self.no)
        # str += ' Standardization parameters' + (' not' if self.Xmeans == None else '') + '
            calculated.'
        if self.trained:
            str += '\n  Network was trained for {} iterations. Final error is
                {}.'.format(self.numberOfIterations,
                                                                                 self.errorTrace[-1])
        else:
            str += ' Network is not trained.'
        return str

    def _standardizeX(self,X):
        return (X - self.Xmeans) / self.Xstds
    def _unstandardizeX(self,Xs):
        return self.Xstds * Xs + self.Xmeans
    def _standardizeT(self,T):
        return (T - self.Tmeans) / self.Tstds
    def _unstandardizeT(self,Ts):
        return self.Tstds * Ts + self.Tmeans

    def _forward_pass(self,X):
        Zprev = X
        Zs = [Zprev]
        for i in range(len(self.nhs)):
            V = self.Vs[i]
            Zprev = np.tanh(np.dot(Zprev,V[1:,:]) + V[0:1,:])
            Zs.append(Zprev)
        Y = np.dot(Zprev, self.W[1:,:]) + self.W[0:1,:]
        return Y, Zs

    def _backward_pass(self,delta,Z):
        dW = np.vstack((np.dot(np.ones((1,delta.shape[0])),delta), np.dot( Z[-1].T, delta)))
        dVs = []
        delta = (1-Z[-1]**2) * np.dot( delta, self.W[1:,:].T)
        for Zi in range(len(self.nhs),0,-1):
            Vi = Zi - 1 # because X is first element of Z
            dV = np.vstack(( np.dot(np.ones((1,delta.shape[0])), delta),
                             np.dot( Z[Zi-1].T, delta)))
            dVs.insert(0,dV)
            delta = np.dot( delta, self.Vs[Vi][1:,:].T) * (1-Z[Zi-1]**2)
        return dVs,dW
```

```
def _pack(self,Vs,W):
    # r = np.hstack([V.flat for V in Vs] + [W.flat])
    # print 'pack',len(Vs), Vs[0].shape, W.shape,r.shape
    return np.hstack([V.flat for V in Vs] + [W.flat])

def _unpack(self,w):
    first = 0
    numInThisLayer = self.ni
    for i in range(len(self.Vs)):
        self.Vs[i][:] =
            w[first:first+(numInThisLayer+1)*self.nhs[i]].reshape((numInThisLayer+1,self.nhs[i]))
        first += (numInThisLayer+1) * self.nhs[i]
        numInThisLayer = self.nhs[i]
    self.W[:] = w[first:].reshape((numInThisLayer+1,self.no))

def pickleDump(self,filename):
    # remove shared memory objects. Can't be pickled
    n = self.iteration.value
    t = self.trained.value
    self.iteration = None
    self.trained = None
    with open(filename,'wb') as fp:
    #    pickle.dump(self,fp)
        cPickle.dump(self,fp)
    self.iteration = mp.Value('i',n)
    self.trained = mp.Value('b',t)
```

3 Experiment

This section explains the details of experiment carried out using linear and non-linear models. I tried to predict number of requests that is 'count' in two different ways as follows.

3.1 Direct Count Prediction

In direct prediction method, I tried to predict number of request that bikeshare system may receive in given hour of given day by using datetime, season, holiday, workingday, weather, temp, atemp, humidity, windspeed. I have not used casual and registered user count because apparently we don't have this information in testing. I have directly predicted total count using initial nine fields. I have divided datetime field in two separate fields, day of the month and hour of the day. So for training I have used 10 fields in total and target is a 'count' field.

Figure 4 and 5 gives predicted versus actual value of total requests for training and testing data with linear model respectively. You can see in the figures that I am getting some negative values that I rounded up to zero. I have used random partitioning of to create testing and training dataset.

Figure 6 and 7 give predicted versus actual total count for training and testing data with neural network respectively. Figure shows that for most of the values non-linear model is conservative in predicting values.

3.2 Indirect Count Prediction

In Indirect approach of predicting total number of request that may come in given hour of given day, I first predicted number of request that may come for casual users and then for registered users. Adding value of casual users and registered users gives total number of users that will request in given hour of given day.
TOTAL COUNT = CASUAL USER COUNT + REGISTERED USERS COUNT
This method gives better results. Figure 8, 9, 10, and 11 gives results for indirect count prediction using non-linear method with 5 hidden units.

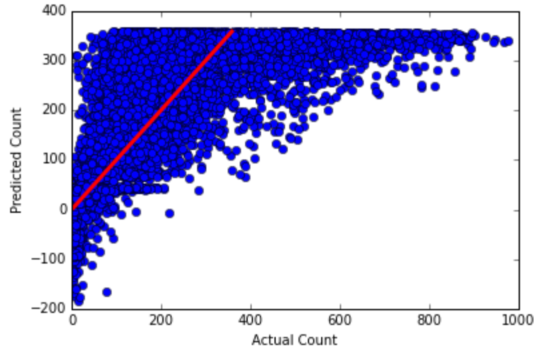

Figure 4: Predicted versus actual total request count for training data with linear model

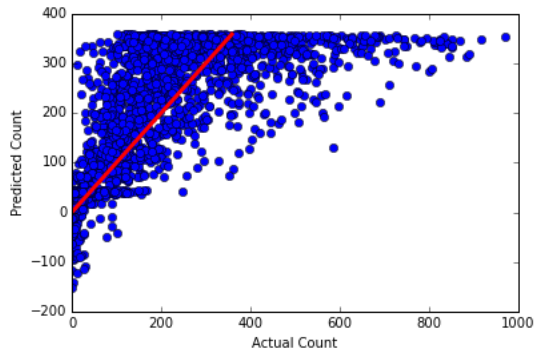

Figure 5: Predicted versus actual total request count for testing data with linear model

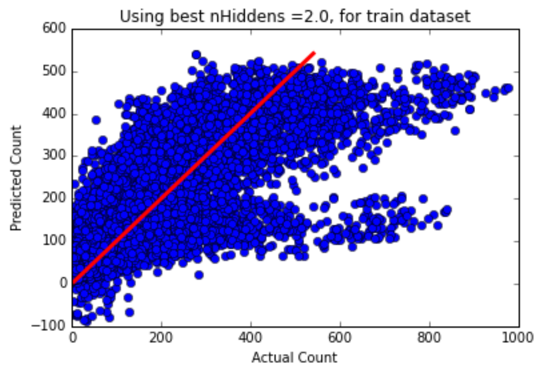

Figure 6: Predicted versus actual total request count for training data with neural network

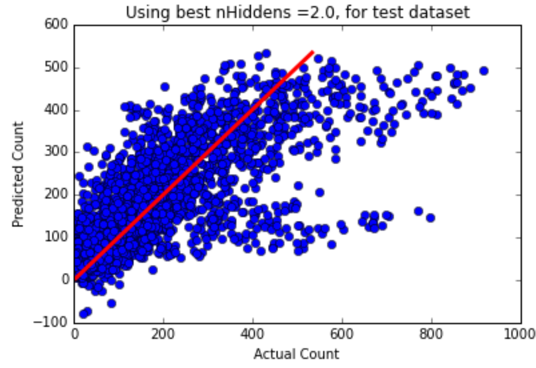

Figure 7: Predicted versus actual total request count for testing data with neural network

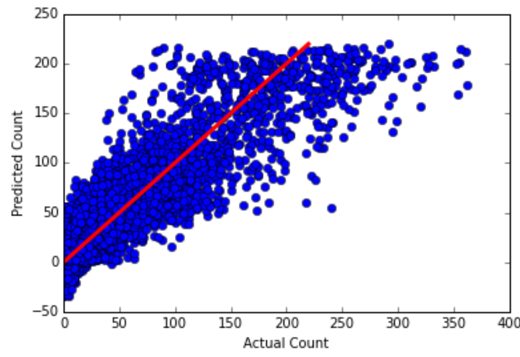

Figure 8: Predicted versus actual casual user's request count for training data with 5 hidden units

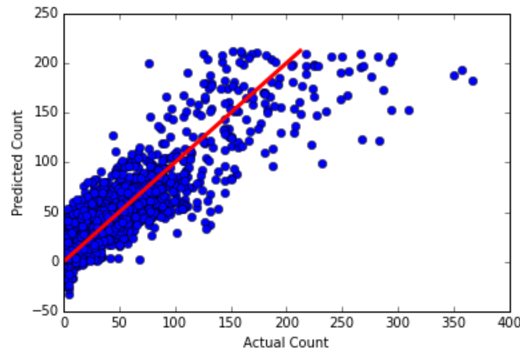

Figure 9: Predicted versus actual casual user's request count for testing data with 5 hidden units

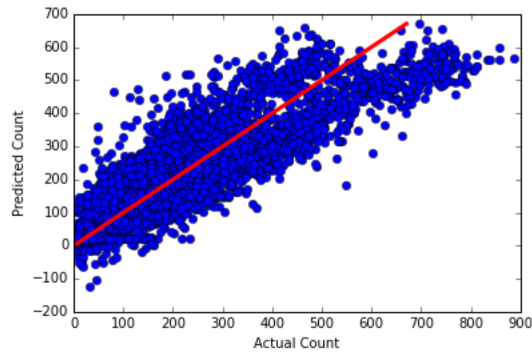

Figure 10: Predicted versus actual registered user's request count for training data with 5 hidden units

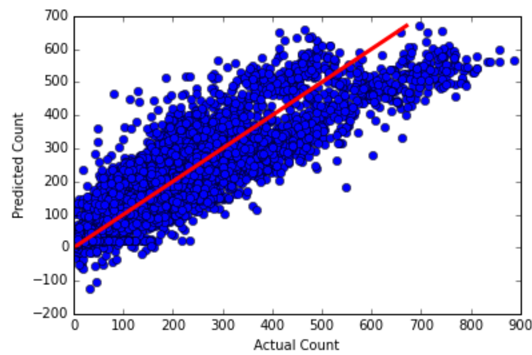

Figure 11: Predicted versus actual registered user's request count for testing data with 5 hidden units

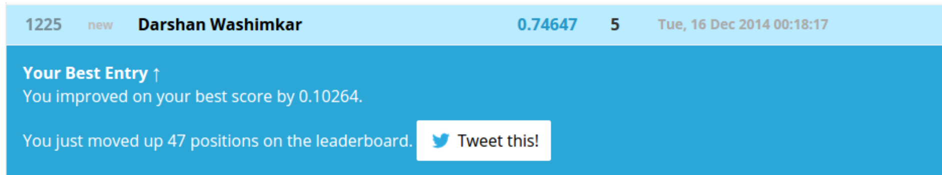

Figure 12: Latest submission result on `https://www.kaggle.com/`

4 Results

I got very interesting result for both models. After using linear model on training data, I got RMSE error as $9.11227 * 10^{-5}$ and for testing data RMSE error was $4.93554 * 10^{-1}$. When I used the non-linear model that is neural network model with 2 as a best hidden unit, I got RMSE error value for training data as $4.05280 * 10^{-05}$ and for testing data I got RMSE erros as $7.89890 * 10^{-1}$. Result suggest that non-linear model performs better to solve this problem. As a further improvement, I adopted indirect prediction approach to predict total count with non-linear model having 5 hidden units. I have submitted all of my results to `https://www.kaggle.com/`. For the latest submission with indirect prediction approach, I was able to improve my result with 0.10264 points and moved 47 positions in tally.

Results shows that non-linear model gives better results than linear model. Non-linear model can perform even better if value of hidden units and hidden layers in fined tunned further.

References

Linear least squares, a. URL `http://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)`.

Artificial neural network, b. URL `http://en.wikipedia.org/wiki/Artificial_neural_network`.

Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Kaggle Inc. Forecast use of a city bikeshare system, November 2014. URL `https://www.kaggle.com/c/bike-sharing-demand`.