University of California, Los Angeles

Math 156

Machine Learning

Neural Networks and Traffic Flow

Authors
Radhika Ahuja
Darragh Glynn
William Legg
Yash Tandon

Instructor
Dr. Stephanie Wang

June 8th, 2020

Introduction

Purpose and Project Goal: With the increasing popularity of ride sharing apps such as Uber and Lyft, it is paramount for these companies to determine trends in taxi cab fares in order to have additional information on the competition. It is also important for ride sharing companies to analyze the trends in how their customers consume their services. With this in mind, we set out to build a model to predict the cost of a taxi ride and determine the times of day which have the greatest price surges.

The Data Set: We used a data set on Kaggle [2] with information about New York City taxi rides to use for our input features which contained the associated fare values for these rides that we could use for our target output. The data set itself contained millions of rows of data but with limited time and computing power, we chose to randomly extract 20,000 points from the data, with 80% being used for training and 20% being used for testing. The following were the input features:

- pickup_datetime - timestamp value indicating when the taxi ride started.
- pickup_longitude - float for longitude coordinate of where the taxi ride started.
- pickup_latitude - float for latitude coordinate of where the taxi ride started.
- dropoff_longitude - float for longitude coordinate of where the taxi ride ended.
- dropoff_latitude - float for latitude coordinate of where the taxi ride ended.
- passenger_count - integer indicating the number of passengers in the taxi ride.

The target output was the following field:

- fare_amount - float dollar amount of the cost of the taxi ride.

Overview of the Methodology: We split this project into four main parts:

1. **Pre-processing the data** - clean and normalize the data entries so they can be effectively fed into models.
2. **Explore Different Models** - research different machine learning models and methodologies before deciding on the optimal methods to use.
3. **Build the Model** - choose and build a model with the necessary parameters (ex. For a Neural Network, determine the number and type of layers as well as the activation functions).
4. **Test the Model and Obtain Insights** - Execute the model on test data and determine the most valuable takeaways from the research.

Model and Optimization Program

Our primary objective is to present the Adam optimization scheme, as developed by the paper by Kingma and Ba [3]. Fix an i.i.d. data set $\mathcal{D} = \{(x_n, t_n) : 1 \leq n \leq k\} \subset \mathbb{R}^m \times \mathbb{R}$. We seek to minimize the error function E induced by maximum likelihood estimation on $\mathcal{D}$:

$$E(w) = \frac{1}{2} \sum_{\mathcal{D}} \left(y(x_n, w) - t_n \right)^2,$$

where y is the network function given by (5.9) in Bishop [1] with activation $h : \mathbb{R}^m \to \mathbb{R}$. Formally, we wish to solve

$$\arg\min_{w \in \mathbb{R}^M} E(w) = \arg\min_{w \in \mathbb{R}^M} \left(\frac{1}{2} \sum_{\mathcal{D}} \left(y(x_n, w) - t_n \right)^2 \right), \tag{1}$$

where $w \in \mathbb{R}^M$ is a (parameter) vector. The Adam scheme solves (1) as follows. Before proceeding, let $\{D_j\}_{j=1}^T$ be a set of randomly (stochastically) generated subsets $D_j \subset \pi(\mathcal{D})$, the projection of $\mathcal{D}$ onto the first m coordinates. We form an associated set $\{E_j\}_{j=1}^T$, where

$$E_j(w_j) = \frac{1}{2} \sum_{D_j} \left(y(x_n, w_j) - t_n \right)^2$$

for suitably re-defined $w \in \mathbb{R}^M$ by $w_j \in \mathbb{R}^M$ (i.e. setting the components $w_{ti} = 0$ for those terms of y of the form $w_{ti} x_i$ in which $x_i \notin D_i$). The Adam algorithm is given below.

Algorithm 1: Adam Optimization

Input: $\alpha \in \mathbb{R}$ (stepsize)
Input: $\beta_1, \beta_2 \in [0, 1]$ (exponential decay rates for the moment estimates)
Input: $\{E_j(w_j)\}_{j=1}^T$ (generated objective functions with parameters w_j)
Input: w_0 (initial parameter vector), $t \leftarrow 0$ (initialize timestep)
$m_0 \leftarrow 0$ (initialize first moment vector)
$v_0 \leftarrow 0$ (initialize second moment vector)
while w_t *has not sufficiently converged* **do**
$t + 1 \leftarrow t$
$g_t \leftarrow \nabla_w E_j(w_{j-1})$ (compute gradients w.r.t. E_j (of timestamp j))
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (update biased second raw moment estimate)
$\overline{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (compute bias-corrected first moment estimate)
$\overline{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (compute bias-corrected second raw moment estimate)
$w_t \leftarrow w_{t-1} - \alpha \cdot \overline{m}_t / (\sqrt{\overline{v}_t} + \epsilon)$ (update parameter vector)
return: w_t (return optimal parameter with t sufficiently updated for convergence)

The following settings are recommended to best ensure the robust convergence of Algorithm 1: $\alpha = 10^{-3}$, $\beta_1 = .9$, $\beta_2 = .999$, and $\epsilon = 10^{-8}$. In the above algorithm, notice that we have abused notation by writing g_t^2 to denote the element-wise product $g_t \odot g_t$.

We will now develop the convergence properties of Adam. Recall that at each timestamp j, our objective is predict the parameter w_j and use to it generate the weights for E_j. Kingma and Ba [3] measure the accuracy of Adam by the regret function $R : \mathbb{N} \to \mathbb{R}$, defined by

$$R(T) = \sum_{t=1}^{T} E_t(w_t) - E_t(w^*).$$

Here, $w^* = \arg\min_{w_t \in \Lambda} \sum_{t=1}^{T} E_t(w_t)$, where $\Lambda = \{w_1, w_2, ..., w_{t-1}\}$. We can also restrict Λ to a proper subset: this may prove useful if a selection algorithm determines certain parameter vectors $w_j \in \mathbb{R}^M$ are not desirable. In what follows, we simplify our notation by $g_t := \nabla E_t(w_t)$, and let $g_{t,i}$ denote the ith element of g_t. We define $g_{1:t,i} \in \mathbb{R}^t$ by $g_{1:t,i} := [g_{1,i}, g_{2,i}, ..., g_{t,i}]$. Finally, let $\gamma := \beta_1^2 \beta_2^{-1/2}$. We have the following result.

Proposition 1. Let $\alpha_t := \alpha/\sqrt{t}$ and $\beta_{1,t} := \beta_1 \lambda^{t-1}$ for $\lambda \in (0, 1)$. Suppose that E_t is such that $\|\nabla E_t(w)\|_2 \leq G$ and $\|\nabla f_t(w)\|_\infty \leq G_\infty$ for constants $G, G_\infty \in \mathbb{R}_{\geq 0}$ and arbitrary $w \in \mathbb{R}^M$. If $\|w_i - w_j\|_2 \leq D$ and $\|w_j - w_i\|_\infty \leq D_\infty$ hold for all $w_k \in \Lambda$ and constants $D, D_\infty \in \mathbb{R}_{\geq 0}$, then

$$R(t) \leq \frac{D^2}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \sqrt{T \overline{v}_{T,i}} + \frac{\alpha(1+\beta_1) G_\infty}{(1-\beta_1)\sqrt{1-\beta_2}(1-\gamma)^2} \sum_{i=1}^{d} \|g_{1:T,i}\|_2 + \sum_{i=1}^{d} \frac{D_\infty^2 G_\infty \sqrt{1-\beta_2}}{2\alpha(1-\beta_1)(1-\lambda)}$$

whenever $\gamma \in [0, 1)$.

Please see the Appendix of Kingma and Ba's paper [3] for the proof of Proposition 1. We can use this bound to show that the average regret $R(T)/T$ converges, as given in the following corollary.

Corollary 1. Suppose that E_t is such that $\|\nabla E_t(w)\|_2 \leq G$ and $\|\nabla f_t(w)\|_\infty \leq G_\infty$ for constants $G, G_\infty \in \mathbb{R}_{\geq 0}$ and arbitrary $w \in \mathbb{R}^M$. If $\|w_i - w_j\|_2 \leq D$ and $\|w_j - w_i\|_\infty \leq D_\infty$ hold for all $w_k \in \Lambda$, then

$$\frac{R(T)}{T} = \mathcal{O}\left(\frac{1}{\sqrt{T}}\right).$$

In particular, we have that

$$\lim_{T \to \infty} \frac{R(T)}{T} = 0.$$

This completes our (albeit brief!) development of the Adam optimization scheme. Please see Kingma and Ba's paper [3] for a further discussion of its empirical properties and extensions.

Implementation

The Data: The data is presented in csv format. Each row represents one ride in a taxi, for example:

FARE AMOUNT	PICKUP TIME	PICKUP LONGITUDE	PICKUP LATITUDE	DROPOFF LONGITUDE	DROPOFF LATITUDE	PASSENGER COUNT
16.9	16:52:16	-74.016048	40.711303	-73.979268	40.782004	1

The Haversine Function: The pick-up and drop-off locations are in degrees latitude and longitude. As we want straight-line distance as an input for our neural network, we use the Haversine function[1] to find the shortest great-circle distance over the Earth's surface between the latitude-longitude coordinates of the pick-up and drop-off, with the answer returned in kilometres.

Cleaning the Data: We clean the data before feeding it into the neural network to catch incorrectly-entered data. We require that each ride has

- Haversine distance < 30km
- cost $< \$75$
- year $\in \{2001, 2002, \ldots, 2019\}$

We also require for simplicity that each ride has passenger count $= 1$, and for general use, the model can be retrained with a passenger count of 2, etc.

Cutting out Manhattan: Upon graphing pick-up and drop-off locations on a map of New York City (see figures below), we notice that the Haversine function will be less accurate if a ride starting on the island of Manhattan does not stay confined to Manhattan. This is because the taxi will *first* have to travel to a bridge and *then* to the drop-off, and the straight-line distance will be an inaccurate measure of the distance.

To make our model more accurate, albeit at the expense of limiting its use, we restrict our data set to rides that begin and end in Manhattan. We identified the points A, B, C, and D as the vertices of a quadrilateral that enclose Manhattan,

$$A = (40.693598, -74.043508) \qquad B = (40.887045, -73.935076)$$
$$C = (40.796631, -73.928354) \qquad D = (40.709264, -73.977696).$$

[1] `https://en.wikipedia.org/wiki/Haversine_formula`

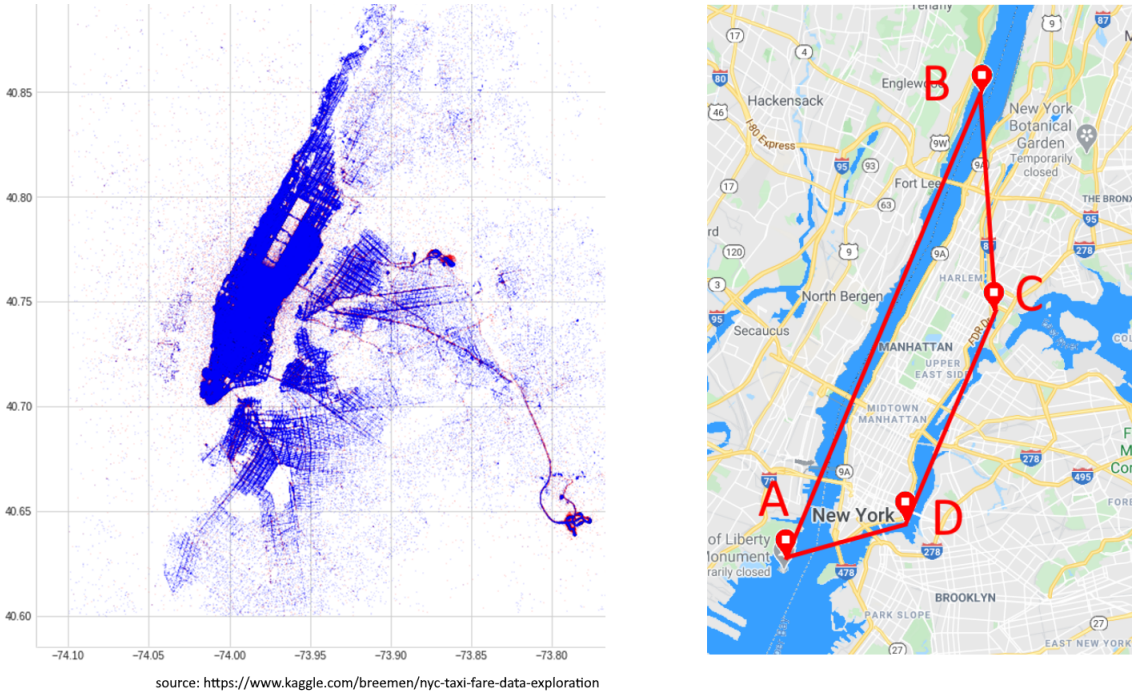


Figure 1: Visualising and Restricting the Data

All points X to the right of the line AB will give $\det(B-A, X-A) > 0$, and all points to the left side of AB will give $\det(B-A, X-A) < 0$. This is because if X is to the right of AB then the matrix $(B-A, X-A)$ is a linear transformation which preserves the orientation of the basis vectors $(1,0)$ and $(0,1)$, and if X is to the left of AB then the matrix $(B-A, X-A)$ is a linear transformation which reverses the orientation of the basis vectors $(1,0)$ and $(0,1)$.

We can easily check the determinants of the relevant matrix for each pick-up or drop-off location X and for each of the four lines of the quadrilateral. If the pick-up and drop-off points of a ride are within the quadrilateral, then the ride is included in the restricted data set.

Compiling the Model: We used the packages Keras and scikitlearn in Python to implement our neural networks. In order to compile the model we need to specify:

- **the optimizer:** the stochastic gradient descent algorithm we wish to use. We used Adam (Adaptive Moment Optimization), a standard choice (see the Math section).
- **the loss function:** this measures how different our model's predicted values are from the actual values at each step of the training process. We used Mean Squared Error.

Comparison of Models: Figures 4 and 5 provide plots of our Basic Simple Linear Model and Basic NN Model (trained and tested on 20,000 points of the unrestricted data set) and our Manhattan

Simple Linear Model and Manhattan NN Model (trained and tested on 20,000 points restricted to Manhattan). Some observations about the graphs of the models:

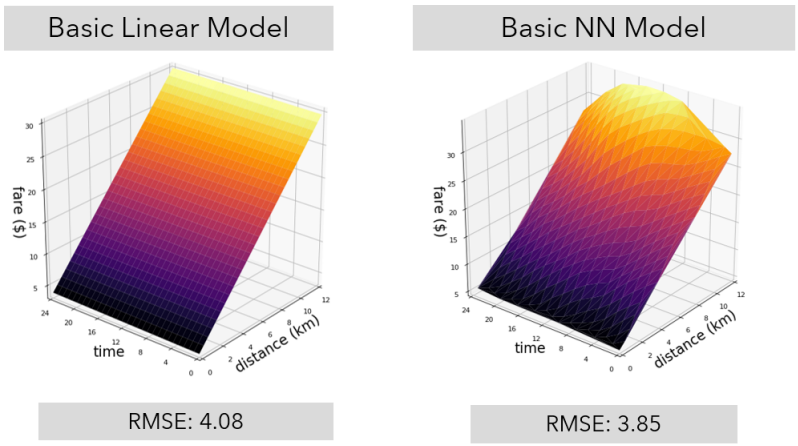


Figure 2: The Models Trained on the Unrestricted Data Set

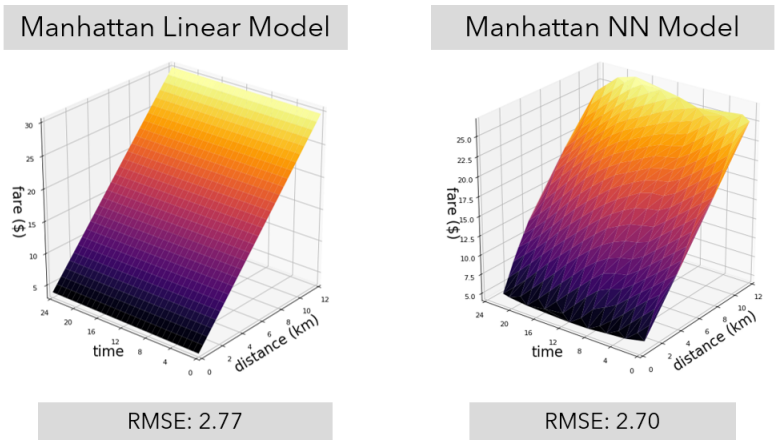


Figure 3: The Models Trained on the Data Set Restricted to Manhattan

- As predicted, the models become more accurate when the input is restricted to Manhattan and the RMSE values drop for both the Simple Linear and Neural Network mordels.
- The Simple Linear Models capture a linear relationship between Haversine distance and fare, but not the more complicated, non-linear dependence of the fare on the time-of-day. The NN Models capture a linear relationship between fare and distance travelled, but their graphs also bulge slightly with the time-of-day to account for the dependence of the fare on the time-of-day. This improvement is reflected by the RMSE value dropping from 4.08 to 3.85 for the basic models and from 2.77 to 2.70 for the Manhattan models.

- The RMSE improves from 3.15 to 2.80 when we restrict our data set to rides that start and end in Manhattan. As mentioned before, the Haversine distance function becomes more accurate for rides that are confined to Manhattan, so we expect this improvement.

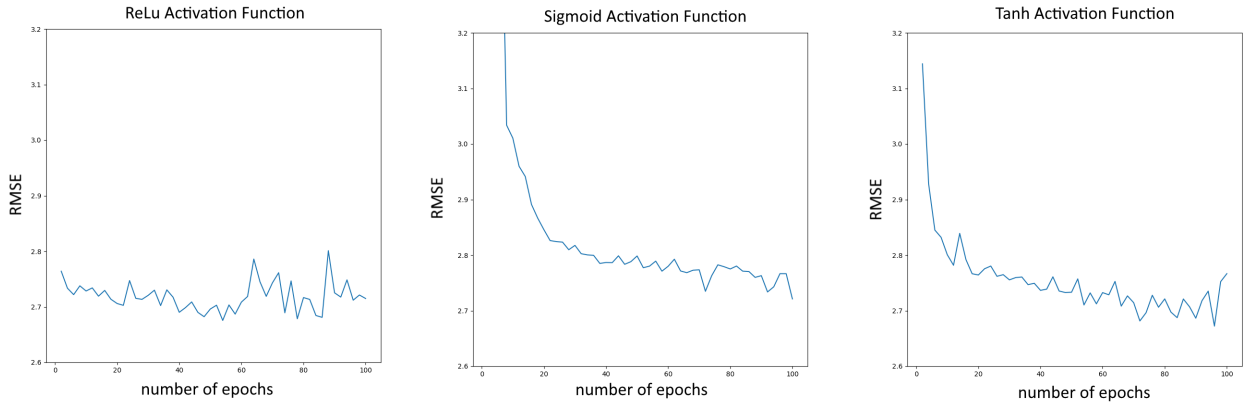


Figure 4: Comparison of Activation Functions

Comparison of Activation Functions: Figure 4 gives a comparison of how the RMSE value developed with the number of epochs for the Relu, tanh and sigmoid activation functions, when training our Manhattan NN Model. The activation functions all behaved similarly, and we had no reason to prefer any one in particular and decided to use a tanh activation function.

Google Maps Driving Distance API: We implemented another method of obtaining distance using the Google Maps API which provides the driving distance between two points, as in the map on the right. This is more accurate than the Haversine function, but due to API limits we were only able to obtain the driving distance for 1,000 data points (unlike the other models that used 20,000). Even so, we trained the model on a subset of the original data set (not restricted to Manhattan) and obtained a RMSE of 3.08, an improvement over the RMSE of 3.20 for the corresponding Basic Neural Network.

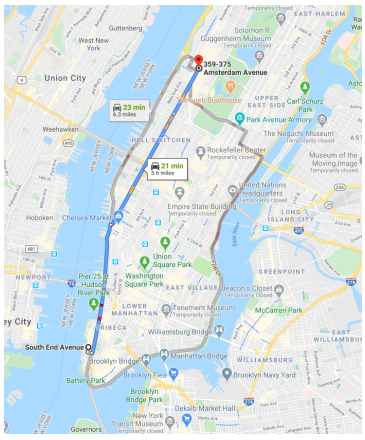
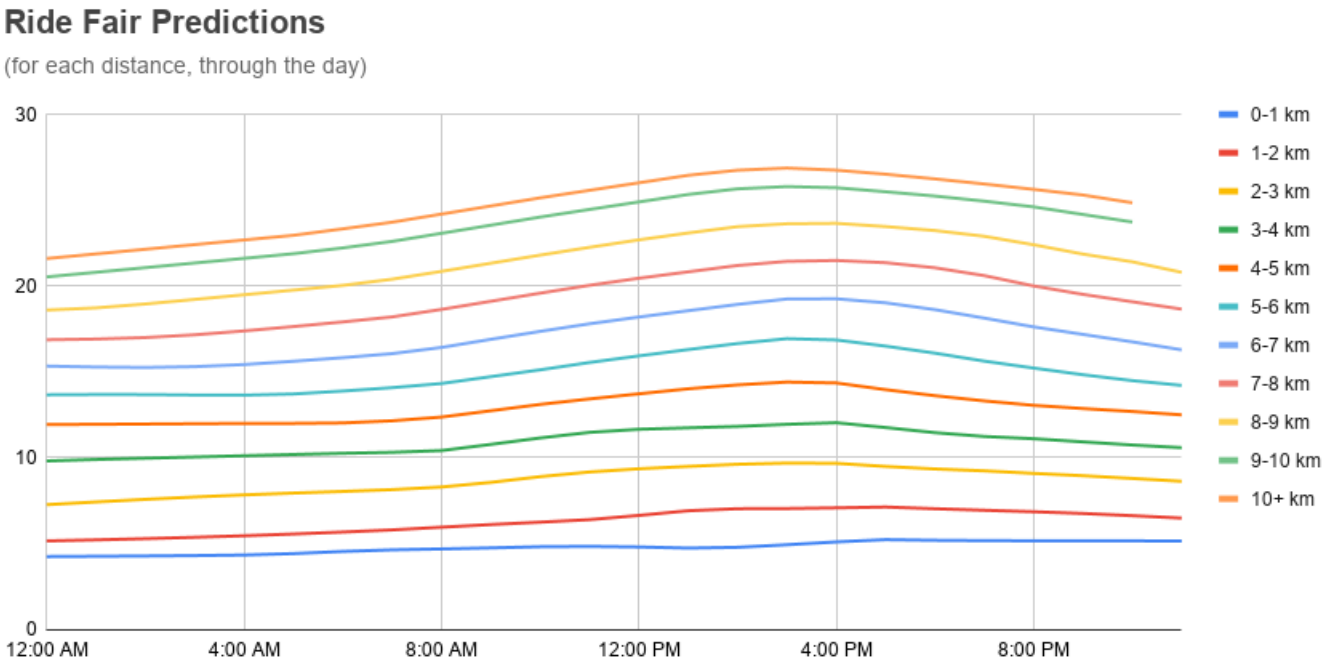


Figure 5: Google Maps Driving Distance

Insights

We can see the following fare distribution based on the model's prediction:



Most notably, we can make the following observations:

- The **prices surge in the middle of the day**, especially between noon and 5pm.
  It is commonly known that rush hour is either earlier in the day, say around 9am or in the evening, so around 6pm. These travellers are probably more regular travellers travelling to work, are more likely to use public transport or their own cars. Thus, mid-day price surges are probably caused more by non-regular travellers or due to unplanned travel.
- The surge is **more pronounced for longer rides** that have a distance of over 5km.
  This is potentially due to a basic pricing model with a base fare and a per km charge. Therefore, as the distance increases, the price increases become more noticeable.

Future Work: Although we obtained some interesting insights, we thought about what ways we can improve upon the model in the future. One approach involves looking into specific data field trends and using that to better pre-process the data, by generating a data field correlation heat map to easily see which fields were most correlated. With this added knowledge, more models with varying data field inputs could be created to find the most optimal specifications for a machine learning model. Examining the test output and determining the source of higher error data points could prove useful in adjusting model parameters and further increasing accuracy.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 9780387310732.

[2] Google Cloud. *New York City Taxi Fare Prediction: Can you predict a rider's taxi fare?* URL: https://www.kaggle.com/c/new-york-city-taxi-fare-prediction.

[3] Jimmy Lei Ba Diederik P. Kingma. "Adam: A Method For Stochastic Optimization". In: *International Conference on Learning Representations* (2015).

Code

```
# GitHub: https://github.com/ahujaradhika/Math-156-Project-1/
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits import mplot3d
from math import radians, cos, sin, asin, sqrt


# We read the data from the csv file into a Pandas data frame
# The pick-up longitude,latitude are in columns 3,4
# The drop-off longitude,latitude are in columns 5,6
# The passenger count is in the column 7
# The time-of-day (in minutes) is in column 12
# The API driving distance (in metres) is in column 14 of API_training2.csv
df = pd.read_csv('train.csv')  # for use with Haversine
df2 = pd.read_csv('API_training2.csv')  # for use with Driving Distance API


# To determine whether a given ride pick-up/drop-off is on the island of
    Manhattan.
def inManhattan(lon,lat):
    ptA = [40.693598, -74.043508]
    ptB = [40.887045, -73.935076]
    ptC = [40.796631, -73.928354]
    ptD = [40.709264, -73.977696]
    # line AB
    a = np.sign((ptB[0] - ptA[0]) * (lon - ptA[1]) - (ptB[1] - ptA[1]) * (lat
    - ptA[0]))
    # line BC
    b = np.sign((ptC[0] - ptB[0]) * (lon - ptB[1]) - (ptC[1] - ptB[1]) * (lat
    - ptB[0]))
    # line CD
    c = np.sign((ptD[0] - ptC[0]) * (lon - ptC[1]) - (ptD[1] - ptC[1]) * (lat
    - ptC[0]))
    # line DA
    d = np.sign((ptA[0] - ptD[0]) * (lon - ptD[1]) - (ptA[1] - ptD[1]) * (lat
```

```
    - ptD[0]))

    if (a == 1 and b == 1 and c == 1 and d == 1):
        return 1
    else:
        return 0


# To determine the straight-line distance across Earth's surface between two points,
# given the latitude and longitude coordinates.
# Source: modified from
# https://gist.github.com/DeanThompson/d5d745eca4e9023c6501#file-haversine-py-L5
def haversine(lon1, lat1, lon2, lat2):
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6369 # The radius of the Earth in New York
    return c * r


# Read the [distance, time-of-day] of up to N rides into A
# and the corresponding fares into b for rides that satisfy:
# Haversine distance < d
# cost < c
# no. passengers = n
# inManhattan=TRUE for pick-up and drop-off points (comment out the line in the if statement
# to include all points)
def readDataHaversine(A, b, N, d, c, n):
    i = 0
    while len(A) < N:
        if (df.iat[i, 7] == n
                and haversine(df.iat[i, 3], df.iat[i, 4], df.iat[i, 5], df.iat[i, 6]) < d
                and df.iat[i, 8] > 0
                and df.iat[i, 1] < c
                and 2000 < df.iat[i, 9] < 2020
```

```
                and inManhattan(df.iat[i, 3], df.iat[i, 4]) == 1
                and inManhattan(df.iat[i, 5], df.iat[i, 6]) == 1
        ):
            A.append([haversine(df.iat[i, 3], df.iat[i, 4], df.iat[i, 5], df.
    iat[i, 6]), df.iat[i, 12]])
            b.append(df.iat[i, 1])
            i += 1
        else:
            i += 1
            continue

    return np.array(A), np.array(b)

def readDataAPI(A, b, N, d, c, n):
    i = 0
    while len(A) < N:
        if (df2.iat[i, 7] == n
                and haversine(df2.iat[i, 3], df2.iat[i, 4], df2.iat[i, 5], df2
    .iat[i, 6]) < d
                and df2.iat[i, 8] > 0
                and df2.iat[i, 1] < c
                and 2000 < df2.iat[i, 9] < 2020
                and inManhattan(df2.iat[i, 3], df2.iat[i, 4]) == 1
                and inManhattan(df2.iat[i, 5], df2.iat[i, 6]) == 1
        ):
            A.append([df2.iat[i,14], df2.iat[i, 12]])
            b.append(df2.iat[i, 1])
            i += 1
        else:
            i += 1
            continue

    return np.array(A), np.array(b)


# We normalize the data with scikitlearn's Standard Scalar before feeding it
    into the NN
def scale(A):
    sc = StandardScaler()
    X = sc.fit_transform(A)
    return X


```

```
# We build the NN using the Keras Sequential model type.
# Adjust the activation functions/number of layers/type of layers as
   appropriate
def buildNN(activation):
    model = Sequential()
    model.add(Dense(32, input_shape=(2,)))
    model.add(Dense(32, activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(1, activation='linear'))
    return model


# We compile and fit the NN with number of epochs = es, batch size = bs
# Adjust the optimizer/loss function
def compileAndFitNN(model, X_train, y_train, es, bs):
    model.compile(optimizer='adam', loss='mse', metrics=['mse'])
    print(model.summary())
    model.fit(X_train, y_train, epochs=es, batch_size=bs)


# The Root Mean Square Accuracy based on the model predictions
def RMSEforNN(model, X_test, y_test):
    y_test_predict = model.predict(X_test)
    RMSE = 0

    for i in range(0, len(X_test)):
        RMSE += (1 / len(X_test)) * pow(y_test_predict[i] - y_test[i], 2)
    RMSE = sqrt(RMSE)

    return RMSE


def RMSEforLinearModel(W, X_test, y_test):

    def f(x, y):
        return W[0] + W[1] * x + W[2] * y

    RMSE = 0

    for i in range(0, len(X_test)):
        RMSE += (1 / len(X_test)) * pow(f(X_test[i][0], X_test[i][1]) - y_test
   [i], 2)
```

```
157
158     RMSE = sqrt(RMSE)
159
160     return RMSE
161
162
163 # Returns the parameters for a maximum likelihood Simple Linear Model for the
        data,
164 # based on Section 3.1 of Pattern Recognition and Machine Learning, by Bishop
165 def simpleLinearModel(A, b):
166     # Phit is the transpose of Phi
167     Phit = np.zeros((3, len(A)), dtype='float')
168     for i in range(0, len(A)):
169         Phit[0][i] = 1
170
171     for i in range(0, len(A)):
172         Phit[1][i] = A[i][0]
173         Phit[2][i] = A[i][1]
174
175     Phi = Phit.transpose()
176     fares = np.zeros((len(A), 1), dtype='float')
177
178     for i in range(0, len(A)):
179         fares[i] = b[i]
180
181     # W is the maximum-likelihood vector of parameters, W = (Phit Phi)^(-1)
        Phit t
182     W = np.linalg.inv(Phit.dot(Phi)).dot(Phit).dot(fares)
183
184     return W
185
186 #
        ##############################################################################
187
188 # We store the data points in A and the corresponding fares in b
189 A = []
190 b = []
191
192 # N is the number of data points to read in
193 N = 20000
194 readDataHaversine(A, b, N, 30, 75, 1)
195
```

```
196
197 # We scale the data before feeding it into the NN
198 X = scale(A)
199 y = b
200
201
202 # We split the data into 80% training, 20% test
203 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
        random_state=42)
204
205
206 # We compile and fit the data, and print the RMSE value
207 model = buildNN('tanh')
208 compileAndFitNN(model, X_train, y_train, 80, 100)
209 print(RMSEforNN(model, X_test, y_test))
210
211
212 # W gives the parameters of the maximum likelihood Simple Linear Model
213 W = simpleLinearModel(X_train, y_train)
214 print(RMSEforLinearModel(W, X_test, y_test))
```