Capstone Project Report
Machine Learning Engineer Nanodegree
Version 1.0

Mike Leske
September, 14 2018

New York City Taxi Fare Prediction

[https://github.com/mikeleske/kaggle-nyc-taxi]

I Definition

Project Overview

New York City is world-famous for its amount of yellow cabs. With the rise of market disrupters like Uber and Lyft it is essential to upfront be aware of the expected ride cost to select the 'best' option in terms of monetary benefit for the user. In August/September 2018 Kaggle hosted a competition together with its partners Google and Coursera aiming to predict regular taxi fares in New York City. [1]

As part of the competition a training dataset of 55 million historic taxi rides (2009 - 2015) has been provided (see section Data Exploration for further dataset information).

Problem Statement

The problem statement of this Kaggle competition is defined as "predicting the fare amount (inclusive of tolls) for a taxi ride in New York City given the pickup and dropoff locations". [1] The predicted fare amount is to be provided as US Dollar and can serve as an estimate for a future taxi rides. A potential use case for such a prediction algorithm is the integration into online navigation services which – in addition to the routing information – intend to provide an estimate of taxi fare costs to a given route inside New York City.

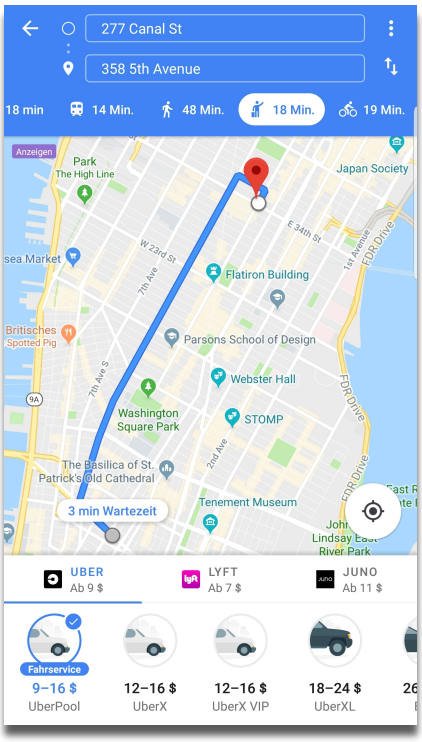

Figure 1 Google Maps providing Uber fare estimation

The outlined problem fits well into the domain of supervised regression problems for which a multitude of potential solutions exist including classical (multivariate) linear regression, decision trees, artificial neural networks and many more.

As part of this capstone project 3 different algorithms will be implemented and compared against each other in respect of accuracy (best result) and efficiency (fastest result).

Metrics

As the problem statement points into the direction of a regression analysis, the competition evaluation is based on the root mean-squared error (RMSE) [2][3]. RMSE is well-suited for this problem statement as it measures the difference between the predicted taxi fares and the real taxi ride fares provided as part of the training dataset. In this sense the taxi ride fare is the regression's dependent variable.

RMSE is calculated by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

A smaller RMSE means the predicted values are close(r) to the real fares of the training set than the predictions of a model with a resulting larger RMSE.

Therefore, the solution for this Kaggle competition seeks to minimize the RMSE.

II Analysis

Data Exploration

The dataset [4] to be used for this competition is provided by Kaggle and its partners Google and Coursera. It contains two essential files:

1. A train.csv file with a training set of 55 million taxi rides and
2. A test.csv file with a test set of approx. 10,000 taxis rides.

For each observation (individual taxi rides) the following parameters are provided:

- key
- pickup_datetime
- pickup_longitude
- pickup_latitude
- dropoff_longitude
- dropoff_latitude
- passenger_count

While the official training set (train.csv) contains the final taxi ride fare, the official test set used for participation in the Kaggle competition does not provide this final insight.

Variable	Dataset "train"	Dataset "test"	Data Type
key	Yes	Yes	String representing pickup_datetime
fare_amount	Yes	No	Recorded fare of taxi ride
pickup_datetime	Yes	Yes	Timestamp of the taxi ride start.
pickup_longitude	Yes	Yes	GPS longitude coordinate where taxi ride started
pickup_latitude	Yes	Yes	GPS latitude coordinate where taxi ride started
dropoff_longitude	Yes	Yes	GPS longitude coordinate where taxi ride ended
dropoff_latitude	Yes	Yes	GPS latitude coordinate where taxi ride ended
passenger_count	Yes	Yes	Number of passengers in the taxi

Table 1 Structure of provided dataset

Note:
Importing the complete training dataset of 55 million observations is very memory demanding. Hence, the following data explorations and model operations will be executed on subsets of the training data, i.e. 10 million observations.

```
df.head()
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	2009-06-15 17:26:21.0000001	4.5	2009-06-15 17:26:00+00:00	-73.844311	40.721319	-73.841610	40.712278	1
1	2010-01-05 16:52:16.0000002	16.9	2010-01-05 16:52:00+00:00	-74.016048	40.711303	-73.979268	40.782004	1
2	2011-08-18 00:35:00.00000049	5.7	2011-08-18 00:35:00+00:00	-73.982738	40.761270	-73.991242	40.750562	2
3	2012-04-21 04:30:42.0000001	7.7	2012-04-21 04:30:00+00:00	-73.987130	40.733143	-73.991567	40.758092	1
4	2010-03-09 07:51:00.000000135	5.3	2010-03-09 07:51:00+00:00	-73.968095	40.768008	-73.956655	40.783762	1

Figure 2 Sample output showing the content of the training dataset

Figure 3 provides an initial statistical overview of the reduced training dataset of 10 million observations. Several observations for data cleaning immediately become visible:

- fare_amount:
  - Negative fares are included in dataset
  - Max fare is much higher than mean and likely represents outliers or extreme rides
- pickup_longitude/pickup_latitude:
  - min & max values represent extreme outliers from mean and 25%/50%/75% percentiles
- dropoff_longitude/dropoff_latitude:
  - count is less than 1 million, representing missing data
  - min & max values represent extreme outliers from mean and 25%/50%/75% percentiles

```
df.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	1.000000e+07	1.000000e+07	1.000000e+07	9.999931e+06	9.999931e+06	1.000000e+07
mean	1.133854e+01	-7.250775e+01	3.991934e+01	-7.250897e+01	3.991913e+01	1.684793e+00
std	9.799930e+00	1.299421e+01	9.322539e+00	1.287532e+01	9.237280e+00	1.323423e+00
min	-1.077500e+02	-3.439245e+03	-3.492264e+03	-3.426601e+03	-3.488080e+03	0.000000e+00
25%	6.000000e+00	-7.399207e+01	4.073491e+01	-7.399139e+01	4.073403e+01	1.000000e+00
50%	8.500000e+00	-7.398181e+01	4.075263e+01	-7.398016e+01	4.075316e+01	1.000000e+00
75%	1.250000e+01	-7.396710e+01	4.076712e+01	-7.396367e+01	4.076810e+01	2.000000e+00
max	1.273310e+03	3.457626e+03	3.344459e+03	3.457622e+03	3.351403e+03	2.080000e+02

Figure 3 Statistical overview over 10 million observations

Exploratory Visualization

This competition aims in predicting most precisely the expected taxi ride fare given GPS coordinates, a date and passenger count. It therefore makes sense to explore the taxi ride fares provided with the training dataset.

Figure 4 visualizes the distribution of the taxi ride fare of 10 million training dataset observations. As indicated already by the initial statistical overview in the previous section, the majority of the all taxi rides result in fares lower than $20. Interestingly, several peaks are visible between $40 and $60.

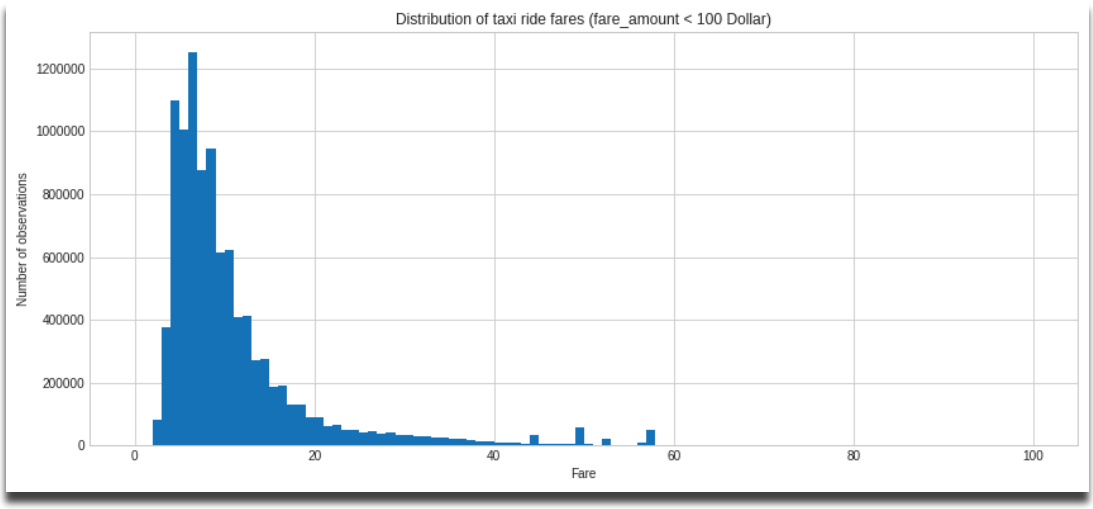

Figure 4 Distribution of taxi ride fares (fare_amount < 100 Dollar)

Figure 5 focuses on the fare_amount distribution, where the announced fare is larger than $20 and less than $100, and confirms strong peaks in the fare_amount distribution in this range. In order to successfully predict taxi ride fares across the test dataset, this specific cost behaviour needs to be correctly learned by the prediction algorithms.

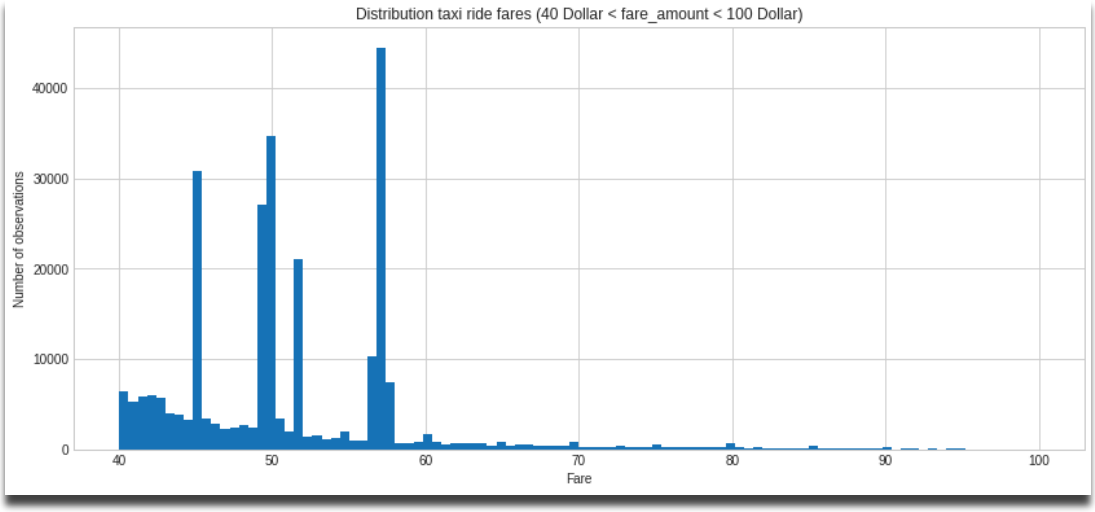

Figure 5 Distribution taxi ride fares (40 Dollar < fare_amount < 100 Dollar)

In order to further explore the specific fare distribution between $40 and $60, new features will be added to the dataset, especially the shortest distance of either pickup or dropoff location to any of the three New York City airports and the distance of the taxi ride given the pickup and dropoff GPS locations.

The GPS coordinates used for each for the 3 New York City airports are:

1. JFK 40.639722 -73.778889
2. EWR 40.6925 -74.168611
3. LGA 40.77725 -73.872611

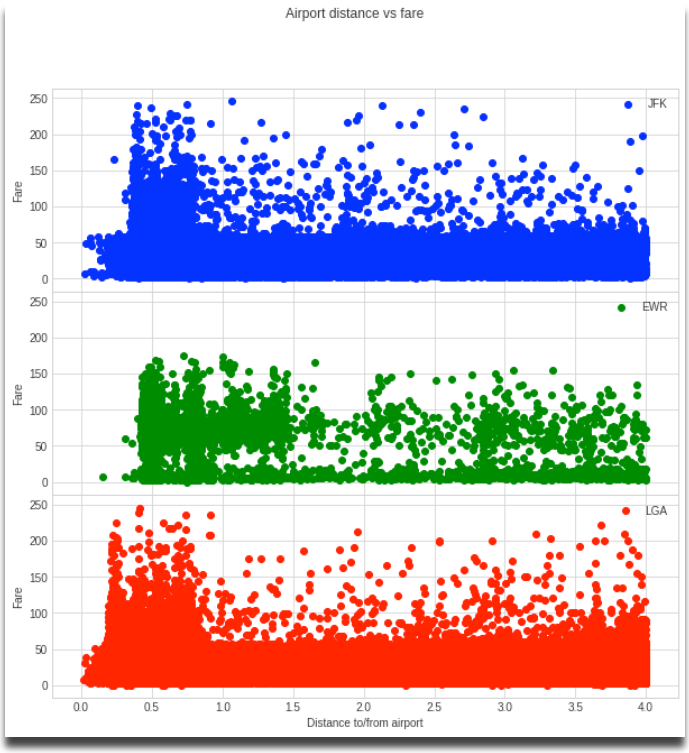

Figure 6 Airport distance vs fare

As can be seen in Figure 6 taxi rides which start or end close to any of the 3 airports correlate to higher taxi rides fare indicating the existence of fixed airport fares. This visualization confirms that whatever prediction methodology will be used has to learn such a dependency between pickup/dropoff airport proximity and expected taxi fare.

Algorithms and Techniques

The New York City Taxi Fare Prediction Kaggle competition presents itself as a classical regression type of problem, which is a well-studied domain with several potential solutions available ranging from classical Machine Learning (ML) algorithms to more sophisticated Deep Learning (DL) architectures based on Artificial Neural Networks (ANN). As described in the Problem Statement section, the provided training data is labelled with typical taxi ride variables like pickup and dropoff coordinates and the number of passengers as well as the "label" in terms of cost of the ride ('fare_amount'). Therefore, the solution to this prediction problem can be targeted using Supervised Learning algorithms.

The following specific algorithms/solutions will be evaluated and compared:

- Multivariate Linear Regression in Python using the scikit-learn library
- Gradient Boosting (XGB) in Python using the xgboost and scikit-learn libraries
- Deep Learning ANN in Python

Multivariate Linear Regression

The first algorithm to predict the taxi ride fares is based on scikit-learn's Linear Regression using Ordinary Least Squares in which a linear model with coefficients $w = (w_1, ..., w_n)$ aims for minimizing residual sum of squares between the actual and the predicted fares. [5]

Considering the non-linearities discovered in the training data in respect to fixed fares for airport transports, this model is expected to roughly match the competition benchmark provided, but not much better.

Out of the total training dataset a chunk of 10 million observations will be used. The outcome of the model is a set of coefficients (equivalent to the number of input variables), which can then be used to predict the taxi fares for observations of the test dataset.

Gradient Boosting (XGB)

Gradient Boosting builds on the foundations of boosting decision trees for classification problems, in which each tree is considered a weak learner, which does not aim modelling all dependencies of the provided dataset. Rather, subsequent weak learners focus on developing new decision trees to handle observations that were not correctly handled by the previous weak learners. The ensemble of the weak learners can then represent a very strong learner.

The Gradient Boosting algorithm represents a generalization of classical boosting algorithms, e.g. AdaBoost, where the objective becomes the minimization of a loss function by creating weak learners sequentially using a gradient descent like procedure. This generalization allows gradient boosting algorithms to use arbitrary differentiable loss functions and therefore making the algorithm suitable also to regression and multiclass classification problems. [6]

Basic Gradient Boosting operations are defined by the loss function to be optimized, weak learner and the additive nature of the model, where sequentially new weak learners are added while previous weak learners remain unchanged.

Improvements to the basic Gradient Boosting operation include:

- Tree Constraints
    - Number of decision trees
    - Depth of the decision trees
    - Number of nodes
    - Number of leaves
    - Number of observations per split
    - Minimum improvement to loss

- Shrinkage
    - Also known as “Learning Rate”
- Random Sampling
    - Also known as “Stochastic Gradient Boosting” where subsequent trees are created based on randomly sampled observations.
    - Stochastic Sampling can be based on observations and dataset columns to be used for a tree creation.
- Penalized Learning
    - L1 and L2 regularization of leaf values in decision trees and help avoiding overfitting.

XGBoost stands for eXtreme Gradient Boosting and is a well-known Gradient Boosting algorithm that recently dominated machine learning and Kaggle competitions especially for structured/tabular data where it has outperformed many classical machine learning algorithms. It implements all the improvements to Basic Gradient Boosting listed above and is optimized to run in a performant manner, e.g. by leveraging parallel CPU cores and cache optimization. [6]

The default options for optimizing the XGBoost operations are specified in the official documentation for the XGBoost project [7].

Deep Learning Artificial Neural Network

Artificial Neural Networks (ANN) are computing environments that are vaguely inspired by communication patterns in biological neural networks. ANNs have been proven successfully in solving both regression, classification and computer vision problems. ANNs have shown exceptional results in discovering hidden pattern in non-linear data that might not immediately be obvious for humans. Hence, solving the New York City Taxi Fare prediction using an ANN seems to be a promising option.

Artificial Neural Networks typically consist out of the following core elements:

- **Input layer** with number of neurons matching the number of input variables
- 1 or multiple **hidden layers** with an arbitrary number of neurons per layer
- **Output layer** providing the prediction given the input

Each neuron from layer n virtually connects to each neuron of layer n+1. These virtual connections are described as “weights”. Neurons in the hidden layers sum up all input signals from neurons of the preceding layer. A layer-specific activation function (typically, ReLu, Tanh or Sigmoid) calculates the neuron’s output to the next layer of the ANN. This way input variables get processed and analysed while the input “signal” is forward-propagated through the ANN.

During the training phase of an ANN the output of the output layer is measured against the target value for the given input and an error is calculated. This error is ultimately back-propagated through the ANN and the weights between neurons of different layers are adjusted to correct for the error. The higher a certain weight is, the more important is the output of a specific neuron for the computation of the receiving neuron. In contrast, once a weight between 2 neurons approaches zero, the output of the sending neuron has little to no relevancy to the computation of the receiving neuron and is ignored.

Many frameworks exist for building Artificial Neural Networks, with the most popular ones being TensorFlow, PyTorch, Caffe, mxnet, etc. Other options for building ANNs include reverting to higher-level frameworks, that provide conceptual APIs for defining Deep Learning environments and silently provide the low-level Deep Learning Framework setup, such as Keras [8] and fast.ai [9].

fast.ai aims to provide an easier to use, more reliable and intuitive way to work with Artificial Neural Networks and is used in Deep Learning courses at University of San Francisco. fast.ai relies on PyTorch as underlying framework and provides several higher-level functionalities that make it an interesting option for this Kaggle competition, such as:

- Learning Rate finder
- Automatic feature-scaling
- Embedding matrices for tabular datasets to discover hidden patterns
- Easy handling of categorical and continuous data inside a dataset
- Easy interface to change number of ANN architecture

Typical algorithm variables that effect the performance of an ANN are:

- Learning Rate for back-propagating errors gradients
- Defining the number of hidden layers
- Defining the number of neurons per layer

Benchmark

When the Kaggle competition was launched a starter solution using a Simple Linear Model was provided to establish a baseline performance. [10] This solution calculates the taxi travel distance from the provided pickup and dropoff longitude/latitude values and subsequently executes numpy matrix multiplication operations.

This Simple Linear Model results in a competition evaluation metric (root-mean-squared-error, see section Metrics) of **$ 5.74**.

The solutions to be implemented as part of this capstone project will be measured and compared against this baseline performance value.

III Methodology

Data Preprocessing

As already indicated in the section for data analysis feature engineering and data cleaning was necessary.

The following features have been engineered for the original training and test dataset:

Name	Comment
"year", "weekday", "hour"	Learn time-based dependencies in the training taxi ride fares. These features were created from the "pickup_datetime" provided with the original dataset.
"distance_miles"	Calculate the taxi ride distance based on the Haversine formula [11] based on the provided GPS coordinates for pickup and dropoff.
"jfk_dist", "ewr_dist", "lga_dist"	Calculate the shortest distance of either pickup or dropoff GPS coordinates to any of the 3 NYC airports. These features are expected to help in modelling the special airport fares.

```
def add_time(df):
    # add time information
    df['year'] = df.pickup_datetime.apply(lambda t: t.year)
    df['weekday'] = df.pickup_datetime.apply(lambda t: t.weekday())
    df['hour'] = df.pickup_datetime.apply(lambda t: t.hour)

def distance(lat1, lon1, lat2, lon2):
    p = 0.017453292519943295 # Pi/180
    a = 0.5 - np.cos((lat2 - lat1) * p)/2 + np.cos(lat1 * p) * np.cos(lat2 * p) * (1 - np.cos((lon2
- lon1) * p)) / 2
    return 0.6213712 * 12742 * np.arcsin(np.sqrt(a)) # 2*R*asin...

def add_travel_vector_features(df):
    # add new column to dataframe with distance in miles
    df['distance_miles'] = distance(df.pickup_latitude, df.pickup_longitude, \
                                    df.dropoff_latitude, df.dropoff_longitude)
    #df['abs_diff_longitude'] = (df.dropoff_longitude - df.pickup_longitude).abs()
    #df['abs_diff_latitude'] = (df.dropoff_latitude - df.pickup_latitude).abs()
    #df['abs_diff_lon_lat'] = (df.abs_diff_longitude + df.abs_diff_latitude)

def add_airport_dist(df):
    """
    Return minumum distance from pickup or dropoff coordinates to each airport.
    JFK: John F. Kennedy International Airport
    EWR: Newark Liberty International Airport
    LGA: LaGuardia Airport
    """
    jfk_coord = (40.639722, -73.778889)
    ewr_coord = (40.6925, -74.168611)
    lga_coord = (40.77725, -73.872611)

    pickup_lat = df['pickup_latitude']
    dropoff_lat = df['dropoff_latitude']
    pickup_lon = df['pickup_longitude']
    dropoff_lon = df['dropoff_longitude']

    pickup_jfk = distance(pickup_lat, pickup_lon, jfk_coord[0], jfk_coord[1])
    dropoff_jfk = distance(jfk_coord[0], jfk_coord[1], dropoff_lat, dropoff_lon)
    pickup_ewr = distance(pickup_lat, pickup_lon, ewr_coord[0], ewr_coord[1])
    dropoff_ewr = distance(ewr_coord[0], ewr_coord[1], dropoff_lat, dropoff_lon)
    pickup_lga = distance(pickup_lat, pickup_lon, lga_coord[0], lga_coord[1])
    dropoff_lga = distance(lga_coord[0], lga_coord[1], dropoff_lat, dropoff_lon)

    df['jfk_dist'] = pd.concat([pickup_jfk, dropoff_jfk], axis=1).min(axis=1)
    df['ewr_dist'] = pd.concat([pickup_ewr, dropoff_ewr], axis=1).min(axis=1)
    df['lga_dist'] = pd.concat([pickup_lga, dropoff_lga], axis=1).min(axis=1)

def add_features(df):
    add_time(df)
    add_travel_vector_features(df)
    add_airport_dist(df)

    return df
```

Specific to the ANN implementation the following variable will be one-hot-encoded to reflect their categorical nature:

- year, weekday, hour

The data cleaning targeted removing either outlier observations or observations with wrong information. Observations with missing variables also have been removed. The algorithms were implemented using chunks of 10 million observation samples, and the number of removed rows due to missing variable was negligibly low.

The following data cleaning was applied to the original training dataset:

Name	Comment
drop	Drop observation due to incomplete information.
"pickup_longitude", "dropoff_longitude"	Limit longitudes to the interval [-75, -73]. This effectively excludes extreme outliers and wrong information.
"pickup_latitude", "dropoff_latitude"	Limit latitudes to the interval [40, 42]. This effectively excludes extreme outliers and wrong information.
"passenger_count"	Limit passenger_count to the interval [0, 6] to match test dataset.
"fare_amount"	Limit fare_amount to the interval [0, 250]. This effectively excludes extreme outliers and wrong information.
"distance_miles"	Limit distance_miles to the interval [0.05, 100]. This effectively excludes extreme outliers and wrong information. It also removes observations with zero mobility between pickup and dropoff, which might indicate wrong data recording.

```
def clean_df(df):
    print('Old size: %d' % len(df))

    # Remove observations with missing values
    df.dropna(how='any', axis='rows', inplace=True)

    # Removing observations with erroneous values
    mask = df['pickup_longitude'].between(-75, -73)
    mask &= df['dropoff_longitude'].between(-75, -73)
    mask &= df['pickup_latitude'].between(40, 42)
    mask &= df['dropoff_latitude'].between(40, 42)
    mask &= df['passenger_count'].between(0, 6)
    mask &= df['fare_amount'].between(0, 250)
    mask &= df['distance_miles'].between(0.05, 100)

    df = df[mask]

    print('New size: %d' % len(df))

    return df
```

After data cleaning the training dataset still contains ~9.63 million observations (instead of 10 million).

```
df_train = add_features(df_train)
df_train = clean_df(df_train)
```

```
Old size: 10000000
New size: 9632477
```

Implementation: Linear Regression

Implementing Ordinary Least Square Multivariate Linear Regression using the scikit-learn library is straightforward and does not allow many variations.

1. 10 million observations from training dataset get imported.

```
df_train = pd.read_csv(TRAIN_PATH, nrows = 10_000_000)
```

2. New features are added and dataset gets cleaned.

```
df_train = add_features(df_train)
df_train = clean_df(df_train)
```

3. Dataset is split into independent data X and dependent variable y ('fare_amount').

4. Split the training dataset in "train" and "test" sets. As the official competition test dataset is small compared to the size of the total training set (only 10,000 observations in test dataset), also the train_test_split was configured to only use 0.1% of the train dataset as test dataset for evaluating the Linear Regression resulting in 9633 observations for validation.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.001)
```

5. Run the Linear Regression algorithm by fitting X_train and y_train to the model. Next evaluate how well the Linear Regression scores by predicting the fares for X_test. As the Linear Regression might predict negative fares, such fares will be set to $0.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import math

model = LinearRegression()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
y_pred[y_pred < 0] = 0

print('Score:', math.sqrt(mean_squared_error(y_test, y_pred)))


Output:
Score: 4.765604755810326
CPU times: user 4.59 s, sys: 1.91 s, total: 6.5 s
Wall time: 6.52 s
```

6. Visualizing the actual fare against predicted fare confirms that the Linear Regression algorithm performs relatively well. The horizontal "scattered lines" at ~$45, $49 and $58 also highlight the existence of the special airport fares. Here the Linear Regression algorithm was not performing too well.

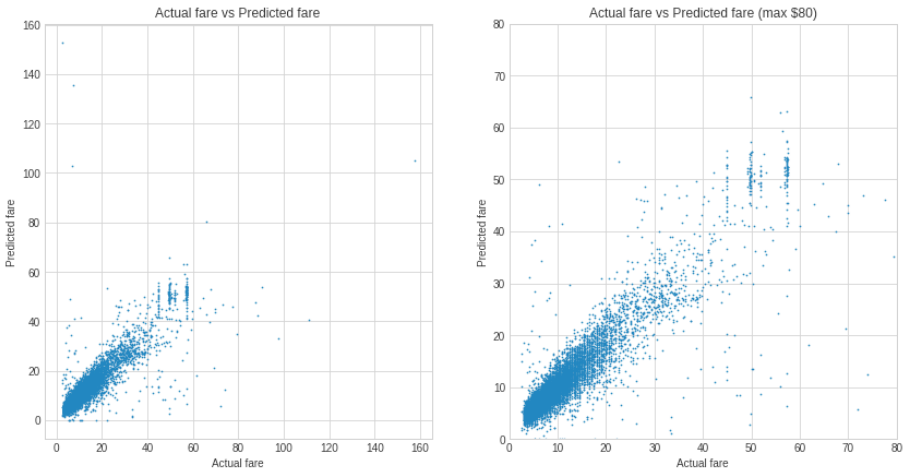

Figure 7 Linear Regression: Actual Fare vs. Predicted Fare

7. The score of predicting the fares of the validation dataset results in RMSE of $4.77. That means the Linear Regression algorithm predicted the fares with an average error of $4.77, which is already significantly lower than the benchmark of $5.74.

8. Finally, the official test dataset must be loaded, features must be added and fares must be predicted. The results are used to participate in the Kaggle competition. The official Kaggle score was calculated as **$4.75**.

```
pred_test = model.predict(X_test)
pred_test[pred_test < 0] = 0

submission = pd.DataFrame(
    {'key': df_test.key, 'fare_amount': pred_test},
    columns = ['key', 'fare_amount'])
submission.to_csv('submission.csv', index = False)
```

Refinement: Linear Regression

Using scikit-learn's Linear Regression algorithm to create a first baseline model was a good start. However, the nature of the problem, especially modelling the dependency of pickup/dropoff proximity to the airports and day/night fares makes it rather unrealistic that strong refinement of the algorithm will dramatically improve the result. Therefore, the Linear Regression will be left as is.

Potential improvements would be:

1. Using k-fold Cross-Validation to improve generalization of the model
2. Use different chunks of 10 million observations.

Implementation: Gradient Boosting

Implementing a Gradient Boosting algorithm requires more steps than a Linear Regression. The Algorithms and Techniques section above already explains the multiple variables of the algorithm itself in order to tune and optimize its performance.

The complete process of working with the XGBoost algorithm included 3 major phases:

1. Baseline performance with default algorithm parameters
2. Multi-dimensional grid search to find optimized algorithm parameters
3. Run algorithm with optimized parameters against dataset

Phases 2 and 3 will be described in the section Refinement: Gradient Boosting. This section continues describing the XGB execution with default parameters.

1. 10 million observations from training dataset get imported.

```
df_train = pd.read_csv(TRAIN_PATH, nrows = 10_000_000)
```

2. New features are added and dataset gets cleaned.

```
df_train = add_features(df_train)
df_train = clean_df(df_train)
```

3. Dataset is split into independent data X and dependent variable y ('fare_amount').

4. Split the training dataset in "train" and "test" sets. The chosen test_size of 5% resulted in a validation dataset of ~500,000 observations. Lastly, convert the train and test sets into XGB matrices.

> **Note:**
> Although the official test set only contains ~10,000 observations and restricting the training validation set to the same size might seem a natural choice, it turned out that XGB training works better, with larger validation sets at the cost of a slight overfitting during training.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05)

matrix_train = xgb.DMatrix(X_train,label=y_train)
matrix_test = xgb.DMatrix(X_test,label=y_test)
```

5. Execute the XGB algorithm with its default parameters and maximum 1000 boosting iterations (tree addition operations). The algorithm is asked to stop early if no improvement was seen for 10 iterations.

```
params = {
    'objective':'reg:linear',
    'eval_metric':'rmse',
    'silent': 1,
    'nthread': 8,
    'n_jobs': 8,
}

model1 = xgb.train(
    params=params,
    dtrain=matrix_train,
    num_boost_round=1000,
    early_stopping_rounds=10,
    evals=[(matrix_test,'test')],
)
```

6. Training completed after 198 iterations, with the best result representing an RMSE of $3.13 in iteration 188. The training RMSE did not improve significantly since iteration 25 with $3.30.

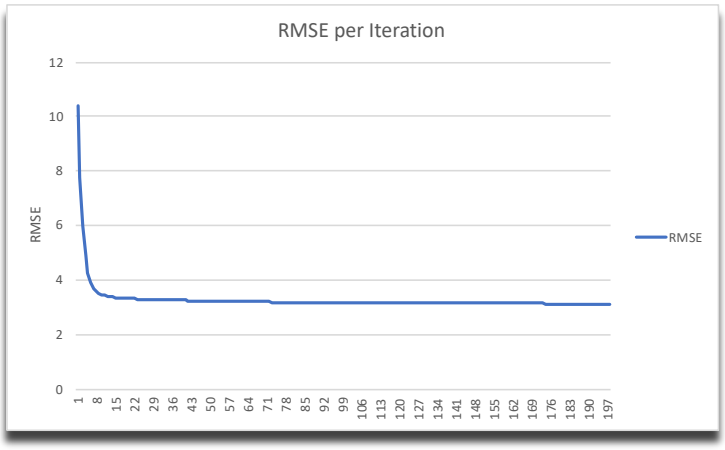

Figure 8 XGBoost: Training progress

Visualizing the actual fare against the predicted fare of the validation set results in a similar graphical representation as the graphic for Linear Regression. It needs to be noted that ~50x more observations were used in this validation set (compared to Linear Regression).

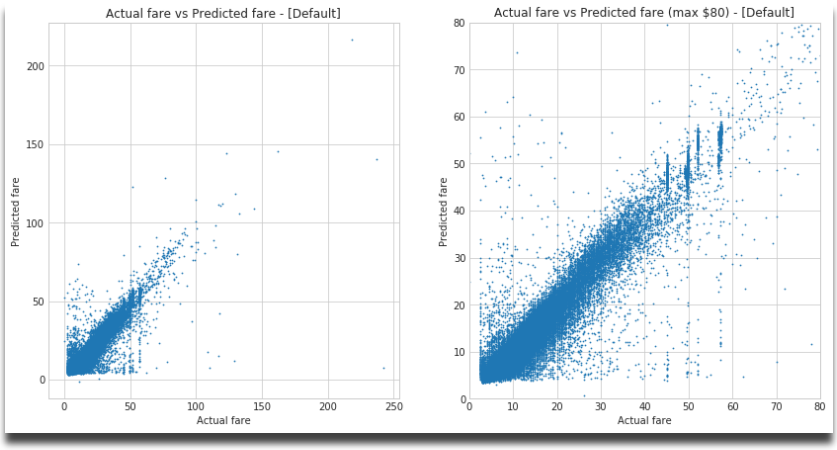

Figure 9 XGBoost: Actual Fare vs. Predicted Fare

7. Next, the official test set is loaded and fares are predicted. The upload of the predicted fare to Kaggle resulted in a score of **$3.73**, which represents a significant improvement over the performance of the Linear Regression.

```
pred_test_default = model1.predict(xgb.DMatrix(X_test), ntree_limit =
model1.best_ntree_limit)

submission_default = pd.DataFrame({
        "key": df_test['key'],
        "fare_amount": pred_test_default.round(2)
})

submission_default.to_csv('taxi_fare_submission_default.csv',index=False)
```

Refinement: Gradient Boosting

Refining a Gradient Boosting algorithm essentially means tuning its multiple parameters on how additional trees are added to the overall model. Section Algorithms and Techniques already introduced the different types of Gradient Boosting optimization variables. For the given problem statement the algorithms optimization search spaces was restricted to the following parameters:

- n_estimators
- max_depth
- learning_rate / eta
- subsample
- colsample_bytree
- comsample_bylevel
- min_weight_child / gamma

Details for any of these algorithm parameters are explained in the XGB documentation [7].

The python scikit-learn library GridSearchCV was used to iterate through the set of potentially optimizable algorithms. “Negative Mean Squared Error” was used of GridSearchCV scoring function, which provides a compatible metric to the actual problem statements’ metric (Root Mean Squared Error). In order to avoid the scale-explosion of multidimensional cross-validation and to allow finding optimized parameters in a reasonable amount of time, a simplified approach was used.

1. Start with a default XGBRegressor object.
2. Set n_estimators to a reasonably low number, such as 250.
3. Restrict training dataset to 500,000 observations to speed up optimization process.
4. For each parameter in (max_depth .. min_child_weight)
    a. Run the algorithm with the parameter values we are interested in
    b. Compare the resulting GridSearchCV scoring values and find the parameter value with the highest score. (Remember: Here, Negative Mean Squared Error is the scoring function which aims for maximization.)
    c. Add the best parameter value to the default model.
    d. Continue GridSearchCV with the next parameter against the slightly improved model.

```
xgb_model = xgb.XGBRegressor(
    objective='reg:linear',
    eval_metric ='rmse',
    nthread = 8,
    n_jobs= 8,
    silent = 1,
)

param_grid = {
    'n_estimators': [250],
    'max_depth': [3, 5, 6, 7, 8, 9, 11],
    'learning_rate': [0.05, 0.06, 0.075, 0.08, 0.1, 0.2, 0.3],
    'subsample': [0.3, 0.4, 0.6, 0.8],
    'colsample_bytree': [0.2, 0.4, 0.6, 0.8],
    'colsample_bylevel': [0.2, 0.4, 0.6, 0.8],
    'min_child_weight': [1, 2, 4, 6, 8]
}
```

```
# Run Grid Search process
fit_params = {
    #'eval_metric': 'rmse',
    'verbose': 0,
    'early_stopping_rounds': 10,
    'eval_set': [(X_test, y_test)]
}

gs_reg = GridSearchCV(
    xgb_model,
    param_grid,
    #n_jobs=1,
    #cv=n_folds,
    fit_params=fit_params,
    scoring='neg_mean_squared_error',
    verbose=1,
    cv=2
)

grid_result = gs_reg.fit(X_train, y_train)

gs_reg.grid_scores_
```

Figure 10 shows the results of the simplified grid search method. After defining a default XGBRegressor objects with 250 estimators, the “max_depth” parameter was optimized by running the algorithm with 7 “max_depth” settings. The best result was reported for max_depth = 7. Before optimizing the “learning_rate” parameter, a new XGBRegressor object with default settings, 250 estimators and a max_depth of 7 was created.

				Default			
n_estimators				250			
max_depth	3 -10.99220	5 -10.48059	6 -10.41637	7 -10.34052	8 -10.37807	9 -10.39652	11 -10.53398
learning_rate	0.05 -10.36742	0.06 -10.38260	0.075 -10.31657	0.08 -10.34036	0.1 -10.34052	0.2 -10.46106	0.3 -10.65314
subsample	0.3 -10.90987	0.4 -10.67439	0.6 -10.32153	0.7 -10.48080	0.8 -10.29832	0.9 -10.28183	
colsample_bytree	0.2 -11.40865	0.4 -10.37820	0.6 -10.20075	0.8 -10.18268			
colsample_bylevel	0.2 -10.71754	0.4 -10.26360	0.6 -10.14766	0.8 -10.20965			
min_child_weight	1 -10.14766	2 -10.14002	4 -10.20543	6 -10.22645	8 -10.32790		

Figure 10 XGBoost: Simplified GridSearch results

The iterative process of finding an optimized XGBoost regressor finally resulted in the following parameters:

```
param_grid = {
    'n_estimators': [250],
    'max_depth': [7],
    'learning_rate': [0.075],
    'subsample': [0.9],
    'colsample_bytree': [0.8],
    'colsample_bylevel': [0.6],
    'min_child_weight': [2]
}
```

Lastly, an optimized XGBRegressor was trained against the same dataset of 10 million observations the default XGBRegressor used as described in the section Implementation: Gradient Boosting. The first 4 steps in preparing the training operation were exactly the same and are not repeated.

5. Execute the XGB algorithm with its **tuned** parameters and maximum of 1000 boosting iterations (tree addition operations). The algorithm is asked to stop early of no improvement was seen for 10 iterations.

```
params = {
    'objective':'reg:linear',
    'eval_metric':'rmse',
    #
    'max_depth': 7,
    'eta':.075,
    'subsample': 0.9,
    'colsample_bytree': 0.8,
    'colsample_bylevel': 0.6,
    'min_child_weight': 2,
    #
    'silent': 1,
    'nthread': 8,
    'n_jobs': 8,
}

model2 = xgb.train(
    params=params,
    dtrain=matrix_train,
    num_boost_round=1000,
    early_stopping_rounds=10,
    evals=[(matrix_test,'test')],
)
```

6. Training completed after 474 iterations, with the best result representing an RMSE of $2.97 in iteration 464. The training RMSE did not improve significantly since iteration 210 with $3.01.

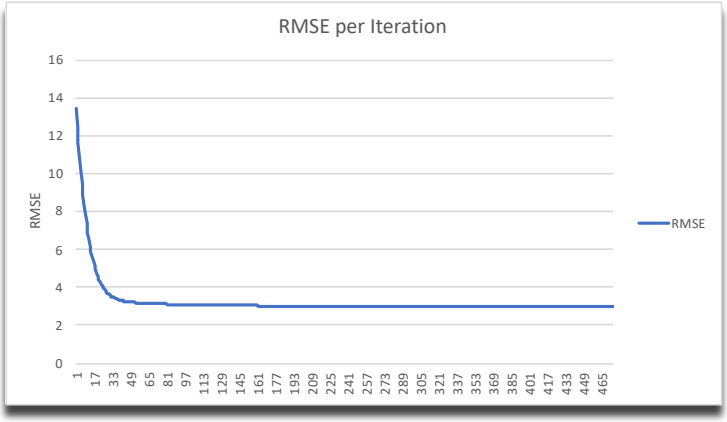

Figure 11 XGBoost (Tuned): Training progress

Visualizing the actual fare against the predicted fare of the validation set results in a similar graphical representation as the graphic for Linear Regression and default XGBoost operation.

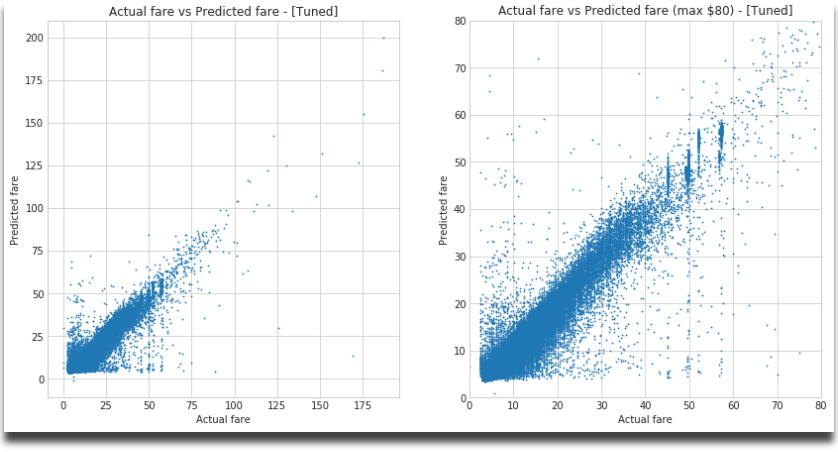

Figure 12 XGBoost (Tuned): Actual Fare vs. Predicted Fare

7. Next, the official test set is loaded and fares are predicted. The upload of the predicted fare to Kaggle resulted in a score of **$3.36**, which represents just another 10% improvement over the default XGBoosting algorithm result of $3.73.

```
pred_test_default = model2.predict(xgb.DMatrix(X_test), ntree_limit =
model2.best_ntree_limit)

submission_default = pd.DataFrame({
        "key": df_test['key'],
        "fare_amount": pred_test_default.round(2)
})

submission_default.to_csv('taxi_fare_submission_tuned.csv',index=False)
```

Implementation: Deep Learning (with fast.ai library)

The third implementation to predict the NYC Taxi fares uses the Deep Learning technology based on an Artificial Neural Network using the fast.ai library [9]. This library acts as a wrapper for the PyTorch Deep Learning framework and targets to ease and simplify the creation on neural networks.

The implementation will run on the Google Colaboratory service, which provides free GPU acceleration for Deep Learning frameworks (Nvidia K80). Like for the XGB implementation, fitting the complete training set into a single model is not possible due to memory constraints. Hence the ANN will be trained on the same chunk of 10M training observations also used for Linear Regression and XGBoosting. This procedure is suboptimal given that the model is subsequently presented with distinct sets of observations, but the overall training dataset of 55M observations seems sufficiently randomized to support this methodology.

The fast.ai library ships with a special class to handle structured, or columnar data, which also implements “entity embeddings” for categorical variables and seems to be a very good match for learning NCY taxi fares. The benefit of “entity embeddings” for categorical variables is provided by the associated “embedding matrices” that can help finding hidden patterns inside the categorical data, e.g. the embeddings can help the algorithm to learn abstract concepts like busy hours – among others. Additional documentation on categorical embeddings can be found on the fast.ai blog [12].

1. Load chunk of 10M training observations, add features and clean data. This step similar to data loading for Linear Regression and XGBoosting and not repeated here. Likewise, the official test data for prediction is loaded.

2. Define categorical and continuous variables from the dataset. Define the dependent variable, here “fare_amount” and set the datatype of categorical variable to “category”. This helps the fast.ai library creating the categorical embedding matrices.

```
cat_vars = ['year', 'weekday', 'hour']

contin_vars = ['pickup_longitude', 'pickup_latitude', 'dropoff_longitude',
               'dropoff_latitude', 'distance_miles', 'jfk_dist', 'ewr_dist',
               'lga_dist']

dep = 'fare_amount'
df_train = df_train[cat_vars+contin_vars+[dep]].copy()
df_test = df_test[cat_vars+contin_vars].copy()

for v in cat_vars:
    df_train[v] = df_train[v].astype('category').cat.as_ordered()
    df_test[v] = df_test[v].astype('category').cat.as_ordered()

apply_cats(df_train, df_test)
```

3. Next, let the fast.ai library prepare the data for processing. This step involves also a scaling operation for each column of the data using a scikit-learn StandardScaler. The scaling operation is subsequently mapped to the test data as well.

```
df_train, y, nas, mapper = proc_df(df_train, 'fare_amount', do_scale=True)
df_test, _, nas, mapper  = proc_df(df_test, do_scale=True, mapper=mapper, na_dict=nas)
```

4. Create a ColumnarDataModel based on the loaded training and test data. Categorical variables are used to build embedding matrices. Batchsize is set to 128.

```
md = ColumnarModelData.from_data_frame('.', val_idx, df_train, y.astype(np.float32),
cat_flds=cat_vars, bs=128,test_df=df_test)

cat_sz = [(c, len(df_train[c].cat.categories)+1) for c in cat_vars]
emb_szs = [(c, min(50, (c+1)//2)) for _,c in cat_sz]
```

The resulting embedding matrices are of the following shapes.

```
[(8, 4), (8, 4), (25, 13)]
```

5. Define a model learner that takes as input variables:
    a. Embedding matrices
    b. Number of continuous variables
    c. Dropout rate for embedding matrices
    d. Output variable(s) (here: 1)
    e. Sizes of hidden neural layers (here: 256, 64, 16)
    f. Dropout rate for hidden neural layers

The fast.ai library internally uses the Adam optimizer, which is not explicitly configured.

```
m = md.get_learner(emb_szs, len(df_train.columns)-len(cat_vars),
                   0., 1, [256,64,16], [0.,0.,0.])
```

Note:
Several different ANN architectures were tried in step e and briefly evaluated. The intention of this "hidden" activity was to find a better-than-random neural network structure.

6. Start training the model for 10 epochs, with a learning_rate of 5e-4 and RMSE as metric.

```
lr = 5e-4
m.fit(lr, 10, metrics=[rmse])
```

Output:

```
HBox(children=(IntProgress(value=0, description='Epoch', max=10), HTML(value='')))

epoch      trn_loss   val_loss   rmse
    0      10.302872  10.194343  3.037156
    1      8.793554   9.987878   3.013298
…
    8      13.664396  9.758162   2.965721
    9      11.75134   9.518649   2.930288
CPU times: user 1h 47min 49s, sys: 21min 21s, total: 2h 9min 11s
Wall time: 2h 2min 49s
```

Figure 13 visualized the training progress of the fast.ai ANN over 10 epochs.

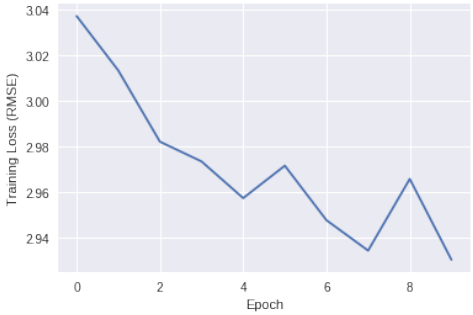

Figure 13 fast.ai ANN (Tuned): Training progress

7. Lastly, predict the taxi fares for the official test set and create a Kaggle submission file. The upload of the predicted fare to Kaggle resulted in a score of **$3.14**, representing another ~7% improvement over the tuned XGBoost result.

```
y_test = m.predict(True)
y_test = y_test.flatten('F').tolist()

df_temp = pd.read_csv('./test.csv')

#Create submission file
submission = pd.DataFrame({
        "key": df_temp['key'],
        "fare_amount": y_test
})

cols=["key","fare_amount"]
submission=submission.reindex(columns=cols)

submission.to_csv('submission_taxi_fare_nn_0.csv',index=False)
```

Refinement: Deep Learning

The execution of the initial ANN training was already fairly optimized. The fast.ai framework provided enhanced features for handling categorical data like “embedding matrices” and a lot of work already went into defining the architecture of the ANN in terms of number of layers, number of neurons per layer, selection of batchsize and the deactivation of dropout.

Subsequent training iterations with different ANN architectures did not result in noteworthy improvements of prediction RMSE. In addition, the free Google Colab GPU resources become unavailable so that further experimentation had to be stopped.

IV Results

Model Evaluation and Validation

Three different models have been implemented, trained against the training dataset and finally predictions for the competitions dataset were inferred from the implementations' model.

For each algorithm several parameters were captured to support comparing them:

- Run-Time (Time needed to complete)
- Result of default algorithm execution for validation and test datasets (only XGB and ANN)
- Result of tuned algorithm execution for validation and test datasets (only XGB and ANN)

	Type	*Training Time (hh:mm:ss)*	*Evaluation Dataset*	*Test Dataset*
Linear Regression	Initial	00:00:06	$4.77	$4.75
Gradient Boosting (XGB)	Default	00:19:56	$3.13	$3.73
	Tuned	00:29:32	$2.97	$3.36
Deep Learning	Initial	02:02:49	$2.94	$3.14
	Tuned	--	--	--

Note:
The number of Deep Learning epochs was arbitrarily set to 10, resulting in a run-time of roughly 2 hours. A similar predictions result *might* be achieved by only using 5 epochs in just 1 hour run-time.

The results provided in the table above represent the RMSE of the competition test dataset predictions. It is immediately obvious that the Deep Learning ANN-based solution produced the best predictions, but it also took the longest time to train the model.

All algorithms beat significantly the performance of the benchmark set initially. XGBoosting and ANN both showed the tendency to slightly overfit, which confirmed the overall structure of the data can be learned. XGBoosting showed a stronger overfitting ratio than the ANN-based solution.

As mentioned earlier, the training dataset of 55M observations was well randomized. There was no temporal or geospatial dependency between the different observations. Hence, relying on a 10M observations training chunk is an appropriate measure in order to work around the memory constraint.

Training the models on different chunks of 10M observations does not result in noteworthy performance deviations, be it positive or negative.

Justification

Implementing the 3 algorithms and their individual improvements was worth the journey. The improvements of the algorithms over the benchmark that was provided are in a significant range between 17% and 45%.

	Type	*Prediction RMSE*	*Improvement over Benchmark*
Benchmark	Initial	$5.74	n/a
Linear Regression	Initial	$4.75	17.25%
Gradient Boosting (XGB)	Default	$3.73	35.02%
	Tuned	$3.36	41.46%
Deep Learning	Initial	$3.14	45.30%
	Tuned	--	--

Figure 14 visualizes that the biggest improvement in prediction RMSE was seen when migrating from the Linear Regression algorithm to the XGBoosting. Deep Learning further improved the taxi fare predictions but the gain was less than expected.

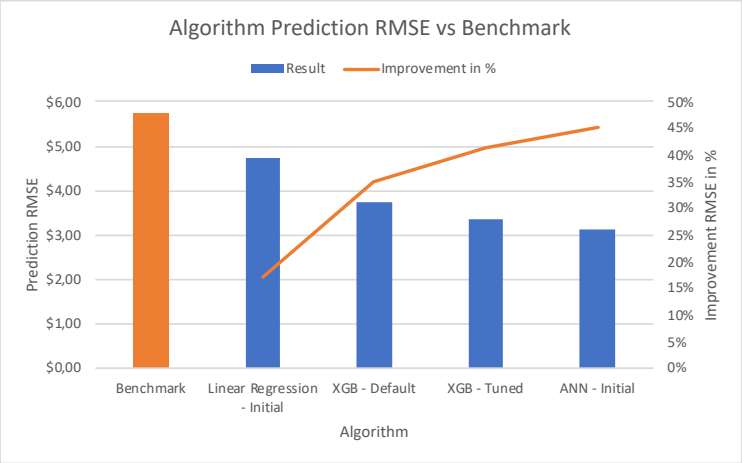

Figure 14 Algorithm Prediction RMSE vs Benchmark

Summarizing, still the Deep Learning ANN-based solution resulted in the best taxi fare predictions and shows more than 45% improvement over the provided benchmark.

V Conclusion

Free-Form Visualization

I did expect that Deep Learning would result in the best prediction results and the fast.ai library did not let me down on this. I also expected large differences in the run-time of the 3 algorithms. However, as explained in the previous section, it was unexpected, that the 4x longer run-time of the ANN only resulted in a ~7% prediction improvement over the XGBoosting solution. This clearly shows one needs to consider a "cost-benefit-trade-off".

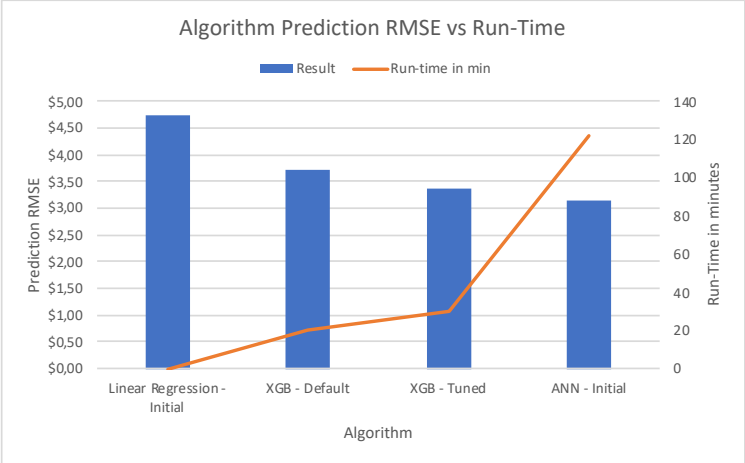

Figure 15 Algorithm Prediction RMSE vs Run-Time

Figure 15 visualizes the mentioned trade-off between prediction results and the cost of getting that level of precision. However, better hardware can easily change this relationship, e.g.:

- More and better GPUs can reduce the ANN run-rime making it more competitive
- XGB can also be training with a GPU (not used here) and that will further and drastically reduce the run-time of the algorithm

Lastly, one would normally select the best performing algorithm (ANN) and try to optimize its training run-time.

Depending on the specific use case, training is only done once and predictions can be subsequently inferred from the model. In a scenario, where the model needs to be re-trained every hour, a 2 hour run-time would not be an option.

Reflection

In this project 3 different solutions have been implemented – all of which differ in complexity and algorithm including a simple Linear Regression, a boosted tree-based solution and a Neural Network.

Integral to finding better-than-baseline predictions was to understand the problem domain and notice that certain pickup and dropoff locations in NYC result in special fares, e.g. airports. Therefore, additional features have been created in order to represent approximate length of the taxi ride in miles and how far/close either pickup or dropoff where to any of the airports. This allowed the algorithms to more easily find patterns in the data than by just processing the raw GPS coordinates.

With exception of the Linear Regression algorithm the other 2 implementations offered many tuneable parameters to modify their performance. For XGBoosting a default solution was tried before the algorithms' operation was optimized. The Deep Learning implementation was already partly optimized be for the initial run.

For each algorithm several parameters were captured to support comparing their performance:

- Run-Time (Time needed to complete)
- Result of default algorithm execution for validation and test datasets (only XGB and ANN)
- Result of tuned algorithm execution for validation and test datasets (only XGB)

Lastly, the before mentioned metrics were compared for a cost-benefit analysis.

Improvement

Certainly, one of the biggest drawbacks of the 3 implementations shown in this study is the inability of training each algorithm on the complete training dataset of 55M observations due to memory constraints. Investing more time to find less memory-intensive representations might improve the situation, but not in the magnitude required to fit the dataset into the free resources provided by Kaggle and Google Colab. Another option would be investing a lot of money into a private compute cluster including multiple GPUs to fit the complete dataset. This, as well, is out of scope for this project.

With respect to optimizing the algorithm a few additional improvements could be implemented for XGBoosting and Deep Learning:

- XGBoosting
  - The XGBoosting model was optimized in respect of the structure of the trees.
  - Additional tuning might be possible by using the L1 and L2 regularization parameters “alpha” and “lambda”.
  - Try to reduce overfitting.
- Deep Learning
  - A core component of Deep Learning is the architecture of the ANN in terms of number of hidden layers and the number of neurons in each layer. Further optimizing this structure can improve the results.
  - “Dropout” functionality might be used inside the ANN. I purposely refused using Dropout as the training dataset already had only a small number of features and risking that either pickup or dropoff coordinates or dropped inside the ANN did not make sense in order to predict the taxi ride fares.
  - Try to reduce overfitting.

With respect to the “cost-benefit-trade-off” between XGBoosting and Deep Learning the following questions are of interest:

- Can we improve the XGBoosting algorithm to match the ANN prediction performance in approximately equal run-time?
- Can the ANN achieve a comparable prediction result in less epochs?
- Can the ANN achieve the XGBoosting result faster than XGBoosting?
- Can a ANN architecture drastically improve the prediction results to justify the longer run-time?
- How does the XGB run-time change when trained on a GPU?

Lastly, completely different algorithms might be tried. The dense nature of the dataset, basically relying solely on GPS coordinates also shows potential for usage by a similarity algorithm of the Approximated Nearest Neighbour class, e.g. annoy.

VI References

[1] https://www.kaggle.com/c/new-york-city-taxi-fare-prediction
[2] https://www.kaggle.com/c/new-york-city-taxi-fare-prediction#evaluation
[3] https://en.wikipedia.org/wiki/Root-mean-square_deviation
[4] https://www.kaggle.com/c/new-york-city-taxi-fare-prediction/data
[5] http://scikit-learn.org/stable/modules/linear_model.html
[6] Book: Machine Learning Mastery - Discover XGBoost With Python!
[7] https://xgboost.readthedocs.io/en/latest/parameter.html
[8] https://keras.io/
[9] http://www.fast.ai/
[10] https://www.kaggle.com/dster/nyc-taxi-fare-starter-kernel-simple-linear-model
[11] https://en.wikipedia.org/wiki/Haversine_formula
[12] http://www.fast.ai/2018/04/29/categorical-embeddings/