Explore and create ML datasets

In this notebook, we will explore data corresponding to taxi rides in New York City to build a Machine Learning model in support of a fare-estimation tool. The idea is to suggest a likely fare to taxi riders so that they are not surprised, and so that they can protest if the charge is much higher than expected.

Learning Objectives

- Access and explore a public BigQuery dataset on NYC Taxi Cab rides
- Visualize your dataset using the Seaborn library
- Inspect and clean-up the dataset for future ML model training
- Create a benchmark to judge future ML model performance off of

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

Let's start with the Python imports that we need.

In [1]:
```
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [2]:
```
from google.cloud import bigquery
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Extract sample data from BigQuery

The dataset that we will use is a BigQuery public dataset. Click on the link, and look at the column names. Switch to the Details tab to verify that the number of records is one billion, and then switch to the Preview tab to look at a few rows.

Let's write a SQL query to pick up interesting fields from the dataset. It's a good idea to get the timestamp in a predictable format.

In [3]:
```
%%bigquery
SELECT
    FORMAT_TIMESTAMP(
        "%Y-%m-%d %H:%M:%S %Z", pickup_datetime) AS pickup_datetime,
    pickup_longitude, pickup_latitude, dropoff_longitude,
    dropoff_latitude, passenger_count, trip_distance, tolls_amount,
    fare_amount, total_amount
FROM
    `nyc-tlc.yellow.trips` # TODO 1
LIMIT 10
```

```
Query complete after 0.01s: 100%|██████████| 2/2 [00:00<00:00, 1047.92query
```

```
/s]
```

Out[3]:

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passe
0	2010-02-05 01:20:05 UTC	-73.979935	40.761105	-73.966230	40.689831	
1	2010-03-07 00:58:45 UTC	-74.001449	40.726071	-73.980448	40.744253	
2	2010-03-05 20:17:51 UTC	-73.863740	40.734245	-73.991364	40.750096	
3	2010-03-29 08:12:38 UTC	-73.993394	40.747158	-73.790150	40.646883	
4	2015-02-22 22:40:31 UTC	-73.937363	40.758041	-73.937386	40.758060	
5	2010-03-14 05:27:23 UTC	-73.993982	40.770577	-73.997214	40.762466	
6	2010-02-04 22:41:28 UTC	-73.991934	40.730339	-73.991934	40.730339	
7	2013-08-15 03:49:56 UTC	-73.937020	40.620175	-73.936452	40.620522	
8	2010-03-02 14:45:23 UTC	-73.973403	40.754323	-73.806456	40.652384	
9	2010-03-11 01:24:14 UTC	-73.990386	40.757301	-74.006484	40.782452	

Let's increase the number of records so that we can do some neat graphs. There is no guarantee about the order in which records are returned, and so no guarantee about which records get returned if we simply increase the LIMIT. To properly sample the dataset, let's use the HASH of the pickup time and return 1 in 100,000 records -- because there are 1 billion records in the data, we should get back approximately 10,000 records if we do this.

We will also store the BigQuery result in a Pandas dataframe named "trips"

In [4]:

```
%%bigquery trips
SELECT
    FORMAT_TIMESTAMP(
        "%Y-%m-%d %H:%M:%S %Z", pickup_datetime) AS pickup_datetime,
    pickup_longitude, pickup_latitude,
    dropoff_longitude, dropoff_latitude,
    passenger_count,
    trip_distance,
    tolls_amount,
    fare_amount,
    total_amount
FROM
    `nyc-tlc.yellow.trips`
WHERE
    ABS(MOD(FARM_FINGERPRINT(CAST(pickup_datetime AS STRING)), 100000)) =
```

```
Query complete after 0.00s: 100%|██████████| 2/2 [00:00<00:00, 885.53query/
s]
Downloading: 100%|██████████| 10789/10789 [00:01<00:00, 8975.49rows/s]
```

In [5]:

```
print(len(trips))
```

```
10789
```

In [6]:

```
# We can slice Pandas dataframes as if they were arrays
trips[:10]
```

Out[6]:

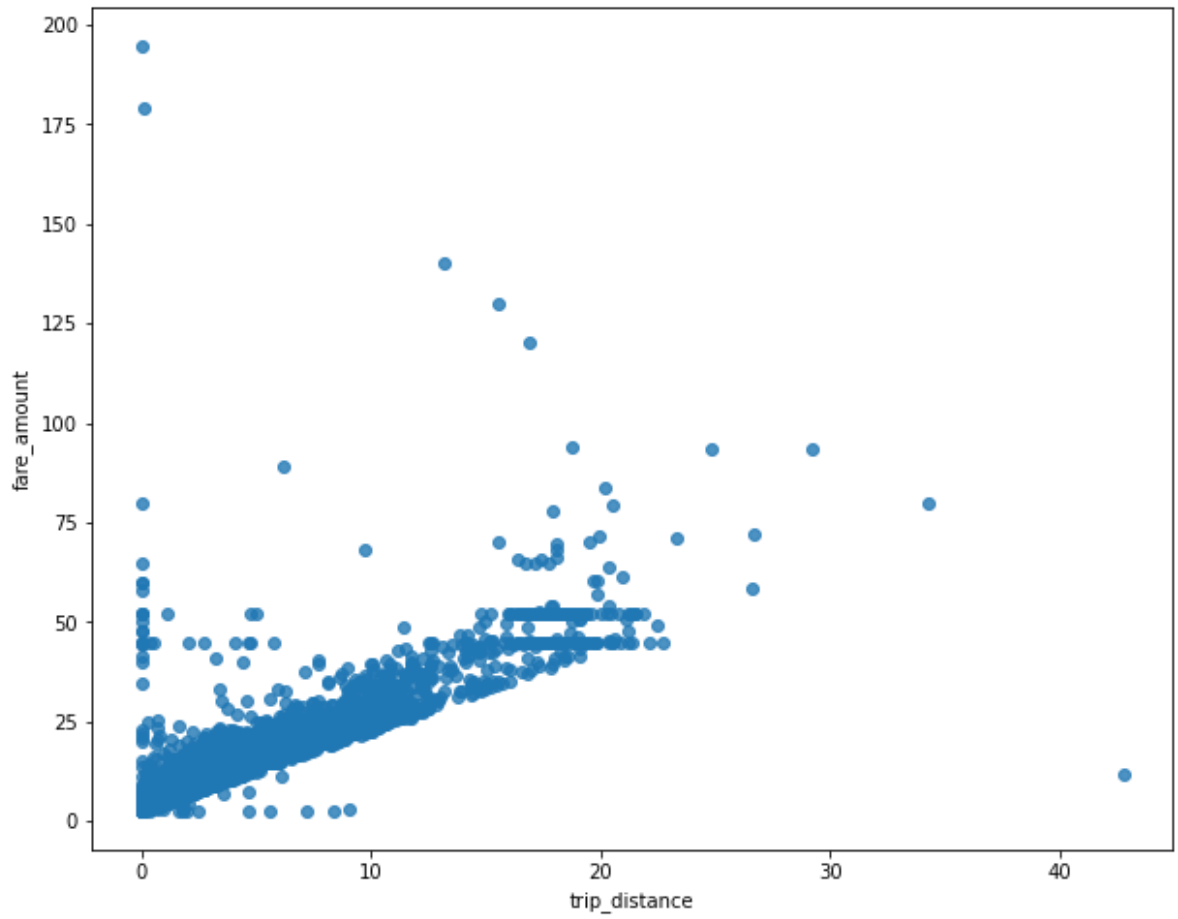
	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passe
0	2014-10-06 15:16:00 UTC	-73.980130	40.760910	-73.861730	40.768330	
1	2014-12-08 21:50:00 UTC	-73.870867	40.773782	-74.003297	40.708215	
2	2010-05-26 16:15:03 UTC	-74.002922	40.714474	-73.978505	40.758280	
3	2009-03-28 20:30:35 UTC	-73.973926	40.757725	-73.981695	40.761591	
4	2009-08-23 23:59:22 UTC	-73.783319	40.648480	-73.893649	40.646566	
5	2013-12-09 15:03:00 UTC	-73.990950	40.749772	-73.870807	40.774070	
6	2012-05-05 22:46:05 UTC	-74.009790	40.712483	-73.959293	40.768908	
7	2010-12-21 13:08:00 UTC	-73.982422	40.739847	-73.981658	40.768732	
8	2012-03-30 18:28:20 UTC	-73.976148	40.776154	-74.010156	40.715113	
9	2014-12-08 21:50:00 UTC	-73.994802	40.720612	-73.949125	40.668893	

Exploring data

Let's explore this dataset and clean it up as necessary. We'll use the Python Seaborn package to visualize graphs and Pandas to do the slicing and filtering.

In [7]:

```
# TODO 2
ax = sns.regplot(
    x="trip_distance", y="fare_amount",
    fit_reg=False, ci=None, truncate=True, data=trips)
ax.figure.set_size_inches(10, 8)
```



Hmm ... do you see something wrong with the data that needs addressing?

It appears that we have a lot of invalid data that is being coded as zero distance and some fare amounts that are definitely illegitimate. Let's remove them from our analysis. We can do this by modifying the BigQuery query to keep only trips longer than zero miles and fare amounts that are at least the minimum cab fare ($2.50).

Note the extra WHERE clauses.

In [8]:

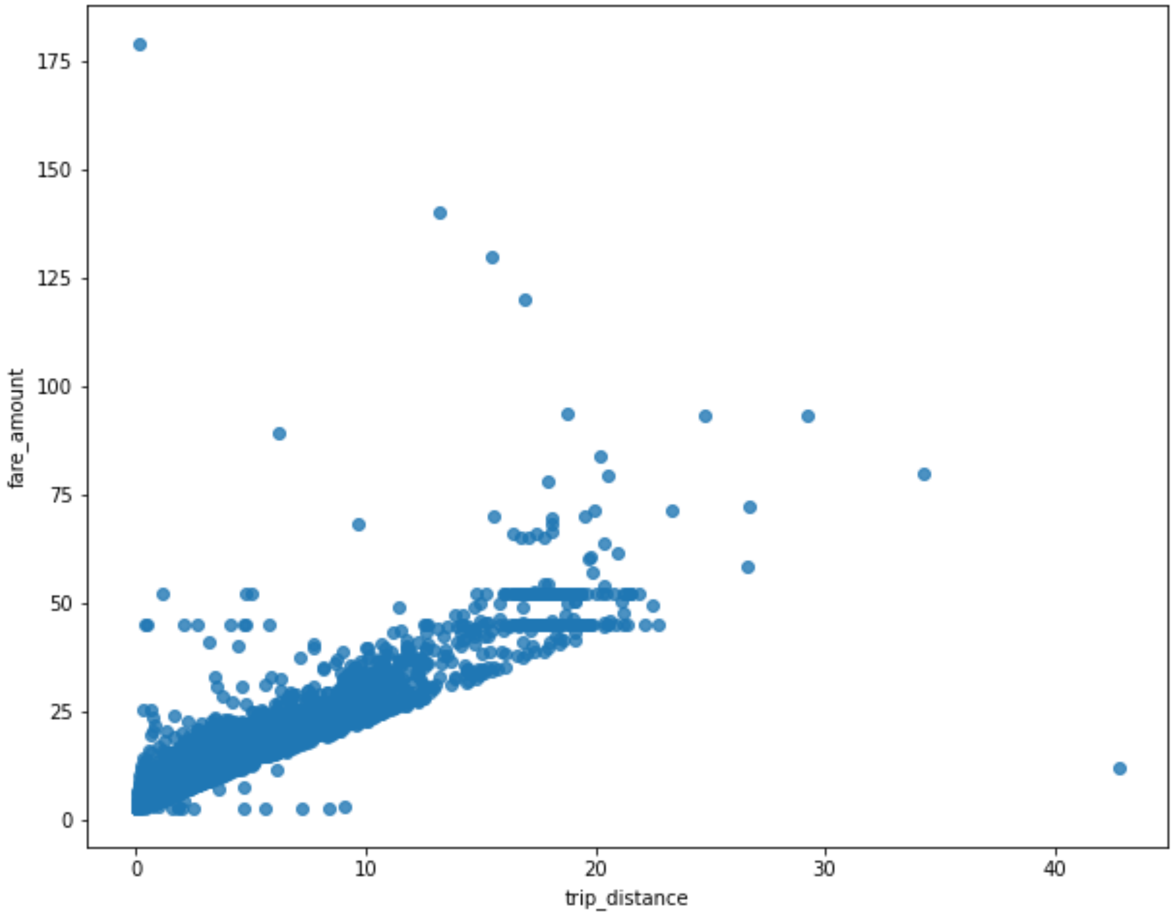
```
%%bigquery trips
SELECT
    FORMAT_TIMESTAMP(
        "%Y-%m-%d %H:%M:%S %Z", pickup_datetime) AS pickup_datetime,
    pickup_longitude, pickup_latitude,
    dropoff_longitude, dropoff_latitude,
    passenger_count,
    trip_distance,
    tolls_amount,
    fare_amount,
    total_amount
FROM
    `nyc-tlc.yellow.trips`
WHERE
    ABS(MOD(FARM_FINGERPRINT(CAST(pickup_datetime AS STRING)), 100000)) = 1
# TODO 3
    AND trip_distance > 0
    AND fare_amount >= 2.5
```

```
Query complete after 0.00s: 100%|██████████| 2/2 [00:00<00:00, 936.54query/
s]
```

In [9]:
```
print(len(trips))
```

10716

In [10]:
```
ax = sns.regplot(
    x="trip_distance", y="fare_amount",
    fit_reg=False, ci=None, truncate=True, data=trips)
ax.figure.set_size_inches(10, 8)
```



What's up with the streaks around 45 dollars and 50 dollars? Those are fixed-amount rides from JFK and La Guardia airports into anywhere in Manhattan, i.e. to be expected. Let's list the data to make sure the values look reasonable.

Let's also examine whether the toll amount is captured in the total amount.

In [11]:
```
tollrides = trips[trips["tolls_amount"] > 0]
tollrides[tollrides["pickup_datetime"] == "2012-02-27 09:19:10 UTC"]
```

Out[11]:

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	pass
25	2012-02-27 09:19:10 UTC	-73.874431	40.774011	-73.983967	40.744082	

In [12]:
```
notollrides = trips[trips["tolls_amount"] == 0]
notollrides[notollrides["pickup_datetime"] == "2012-02-27 09:19:10 UTC"]
```

Out[12]:

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	p

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	p
59	2012-02-27 09:19:10 UTC	-73.972311	40.753067	-73.957389	40.817824	
7789	2012-02-27 09:19:10 UTC	-73.987582	40.725468	-74.016628	40.715534	
10537	2012-02-27	-74.015483	40.715279	-73.998045	40.756273	

Looking at a few samples above, it should be clear that the total amount reflects fare amount, toll and tip somewhat arbitrarily -- this is because when customers pay cash, the tip is not known. So, we'll use the sum of fare_amount + tolls_amount as what needs to be predicted. Tips are discretionary and do not have to be included in our fare estimation tool.

Let's also look at the distribution of values within the columns.

In [13]:

```python
trips.describe()
```

Out[13]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	10716.000000	10716.000000	10716.000000	10716.000000	10716.000000
mean	-72.602192	40.002372	-72.594838	40.002052	1.650056
std	9.982373	5.474670	10.004324	5.474648	1.283577
min	-74.258183	0.000000	-74.260472	0.000000	0.000000
25%	-73.992153	40.735936	-73.991566	40.734310	1.000000
50%	-73.981851	40.753264	-73.980373	40.752956	1.000000
75%	-73.967400	40.767340	-73.964142	40.767510	2.000000
max	0.000000	41.366138	0.000000	41.366138	6.000000

Hmm ... The min, max of longitude look strange.

Finally, let's actually look at the start and end of a few of the trips.

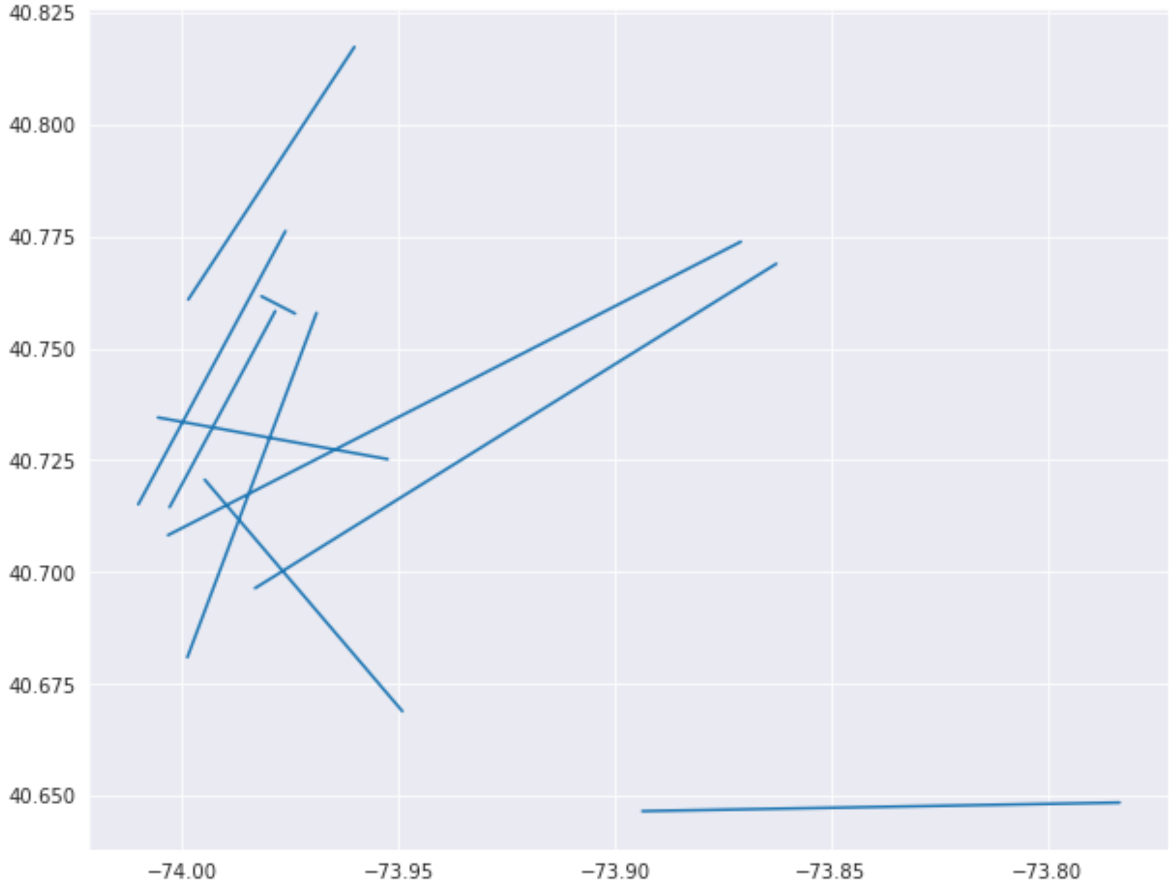
In [14]:

```python
def showrides(df, numlines):
    lats = []
    lons = []
    for iter, row in df[:numlines].iterrows():
        lons.append(row["pickup_longitude"])
        lons.append(row["dropoff_longitude"])
        lons.append(None)
        lats.append(row["pickup_latitude"])
        lats.append(row["dropoff_latitude"])
        lats.append(None)

    sns.set_style("darkgrid")
    plt.figure(figsize=(10, 8))
    plt.plot(lons, lats)
```

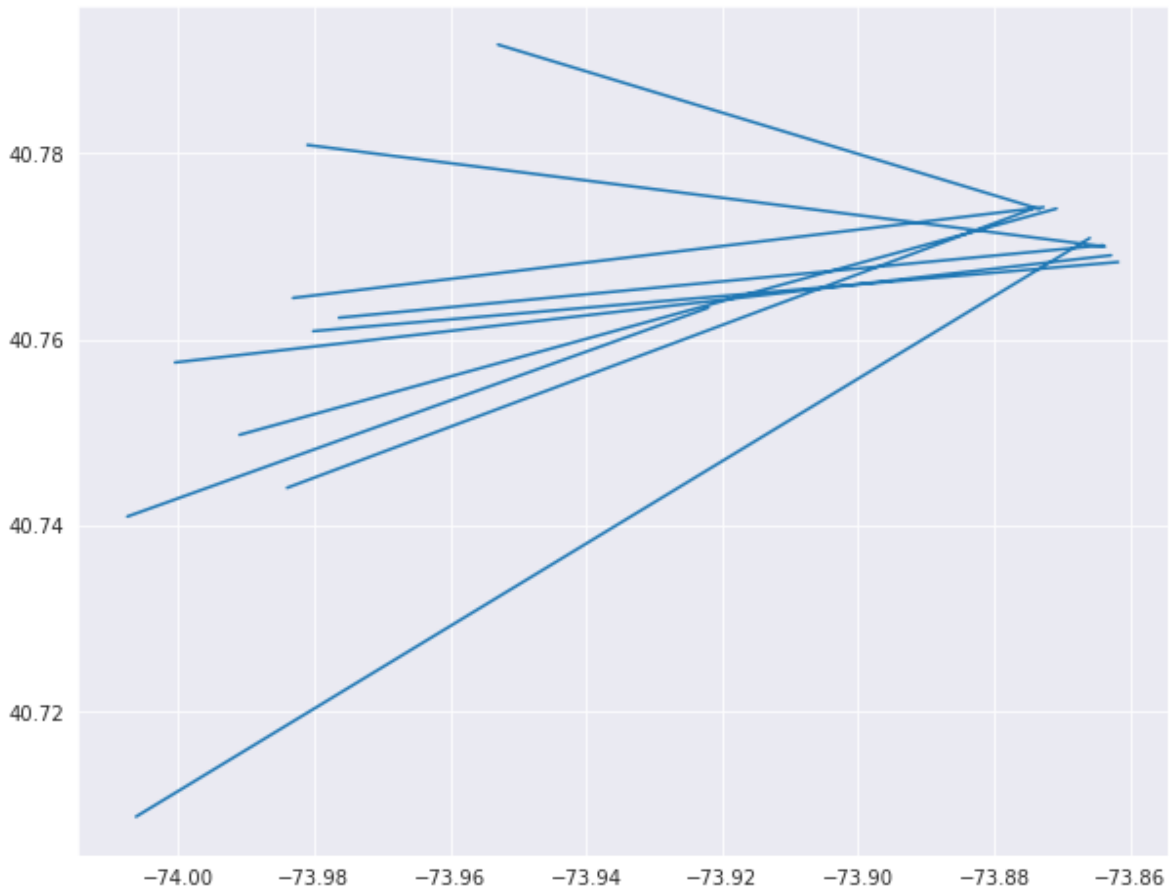
In [15]:

```python
showrides(notollrides, 10)
```



In [16]:

```
showrides(tollrides, 10)
```



As you'd expect, rides that involve a toll are longer than the typical ride.

Quality control and other preprocessing

We need to do some clean-up of the data:

1. New York city longitudes are around -74 and latitudes are around 41.
2. We shouldn't have zero passengers.
3. Clean up the total_amount column to reflect only fare_amount and tolls_amount, and then remove those two columns.
4. Before the ride starts, we'll know the pickup and dropoff locations, but not the trip distance (that depends on the route taken), so remove it from the ML dataset
5. Discard the timestamp

We could do preprocessing in BigQuery, similar to how we removed the zero-distance rides, but just to show you another option, let's do this in Python. In production, we'll have to carry out the same preprocessing on the real-time input data.

This sort of preprocessing of input data is quite common in ML, especially if the quality-control is dynamic.

In [17]:

```
def preprocess(trips_in):
    trips = trips_in.copy(deep=True)
    trips.fare_amount = trips.fare_amount + trips.tolls_amount
    del trips["tolls_amount"]
    del trips["total_amount"]
    del trips["trip_distance"]  # we won't know this in advance!

    qc = np.all([
        trips["pickup_longitude"] > -78,
        trips["pickup_longitude"] < -70,
        trips["dropoff_longitude"] > -78,
        trips["dropoff_longitude"] < -70,
        trips["pickup_latitude"] > 37,
        trips["pickup_latitude"] < 45,
        trips["dropoff_latitude"] > 37,
        trips["dropoff_latitude"] < 45,
        trips["passenger_count"] > 0
    ], axis=0)

    return trips[qc]

tripsqc = preprocess(trips)
tripsqc.describe()
```

Out[17]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	10476.000000	10476.000000	10476.000000	10476.000000	10476.000000
mean	-73.975206	40.751526	-73.974373	40.751199	1.653303
std	0.038547	0.029187	0.039086	0.033147	1.278827
min	-74.258183	40.452290	-74.260472	40.417750	1.000000
25%	-73.992336	40.737600	-73.991739	40.735904	1.000000
50%	-73.982090	40.754020	-73.980780	40.753597	1.000000
75%	-73.968517	40.767774	-73.965851	40.767921	2.000000

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count

The quality control has removed about 300 rows (11400 - 11101) or about 3% of the data. This seems reasonable.

Let's move on to creating the ML datasets.

Create ML datasets

Let's split the QCed data randomly into training, validation and test sets. Note that this is not the entire data. We have 1 billion taxicab rides. This is just splitting the 10,000 rides to show you how it's done on smaller datasets. In reality, we'll have to do it on all 1 billion rides and this won't scale.

In [18]:

```
shuffled = tripsqc.sample(frac=1)
trainsize = int(len(shuffled["fare_amount"]) * 0.70)
validsize = int(len(shuffled["fare_amount"]) * 0.15)

df_train = shuffled.iloc[:trainsize, :]
df_valid = shuffled.iloc[trainsize:(trainsize + validsize), :]
df_test = shuffled.iloc[(trainsize + validsize):, :]
```

In [19]:

```
df_train.head(n=1)
```

Out[19]:

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	pa
1920	2010-04-29 12:28:00 UTC	-73.791475	40.851292	-73.942248	40.803482	

In [20]:

```
df_train.describe()
```

Out[20]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	f
count	7333.000000	7333.000000	7333.000000	7333.000000	7333.000000	
mean	-73.975204	40.751559	-73.974490	40.751315	1.654166	
std	0.038963	0.029417	0.039060	0.033418	1.278626	
min	-74.116582	40.626968	-74.182503	40.561076	1.000000	
25%	-73.992297	40.737810	-73.991645	40.736079	1.000000	
50%	-73.982082	40.754207	-73.980577	40.753571	1.000000	
75%	-73.968755	40.767831	-73.965936	40.767962	2.000000	
max	-73.137393	41.366138	-73.137393	41.366138	6.000000	

In [21]:

```
df_valid.describe()
```

Out[21]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	f
count	1571.000000	1571.000000	1571.000000	1571.000000	1571.000000	
mean	-73.974806	40.751142	-73.974047	40.751899	1.678549	

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
std	0.034250	0.025644	0.041503	0.034113	1.293459
min	-74.187541	40.641300	-74.187541	40.569997	1.000000
25%	-73.991909	40.737800	-73.991970	40.736711	1.000000
50%	-73.981805	40.753700	-73.981017	40.754137	1.000000
75%	-73.967285	40.767097	-73.965681	40.768262	2.000000

In [22]:

```
df_test.describe()
```

Out[22]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	1572.000000	1572.000000	1572.000000	1572.000000	1572.000000
mean	-73.975616	40.751752	-73.974153	40.749956	1.624046
std	0.040618	0.031372	0.036665	0.030810	1.265216
min	-74.258183	40.452290	-74.260472	40.417750	1.000000
25%	-73.993158	40.736723	-73.991852	40.734644	1.000000
50%	-73.982383	40.753513	-73.981460	40.753392	1.000000
75%	-73.968629	40.767668	-73.965238	40.766984	2.000000
max	-73.137393	41.366138	-73.711521	40.864124	6.000000

Let's write out the three dataframes to appropriately named csv files. We can use these csv files for local training (recall that these files represent only 1/100,000 of the full dataset) just to verify our code works, before we run it on all the data.

In [23]:

```
def to_csv(df, filename):
    outdf = df.copy(deep=False)
    outdf.loc[:, "key"] = np.arange(0, len(outdf))  # rownumber as key
    # Reorder columns so that target is first column
    cols = outdf.columns.tolist()
    cols.remove("fare_amount")
    cols.insert(0, "fare_amount")
    print (cols)  # new order of columns
    outdf = outdf[cols]
    outdf.to_csv(filename, header=False, index_label=False, index=False)

to_csv(df_train, "taxi-train.csv")
to_csv(df_valid, "taxi-valid.csv")
to_csv(df_test, "taxi-test.csv")
```

```
['fare_amount', 'pickup_datetime', 'pickup_longitude', 'pickup_latitude', '
dropoff_longitude', 'dropoff_latitude', 'passenger_count', 'key']
['fare_amount', 'pickup_datetime', 'pickup_longitude', 'pickup_latitude', '
dropoff_longitude', 'dropoff_latitude', 'passenger_count', 'key']
['fare_amount', 'pickup_datetime', 'pickup_longitude', 'pickup_latitude', '
dropoff_longitude', 'dropoff_latitude', 'passenger_count', 'key']
```

In [24]:

```
!head -10 taxi-valid.csv
```

```
5.5,2014-10-06 15:16:00 UTC,-73.994408,40.745935,-73.993313,40.736148,1,0
5.5,2013-04-18 08:48:00 UTC,-73.997877,40.740967,-73.99855,40.735062,1,1
5.7,2009-05-27 20:37:00 UTC,-73.979665,40.776338,-73.992835,40.757935,1,2
7.5,2012-11-19 17:41:00 UTC,-73.989115,40.773707,-73.972587,40.786587,1,3
```

```
45.0,2011-10-10 18:21:05 UTC,-73.984449,40.768208,-73.776345,40.645209,1,4
9.5,2015-02-01 02:19:26 UTC,-74.00153350830078,40.73094177246094,-73.985351
5625,40.76145935058594,3,5
11.5,2013-09-17 14:10:35 UTC,-73.958142,40.773595,-73.971527,40.761502,1,6
7.7,2011-12-03 22:51:13 UTC,-73.981466,40.741465,-73.987959,40.724034,3,7
12.1,2012-03-28 18:57:29 UTC,-73.987175,40.738556,-73.982384,40.739991,1,8
8.0,2015-05-16 02:01:03 UTC,-73.97833251953125,40.78310775756836,-73.954048
15673828,40.78716278076172,2,9
```

Verify that datasets exist

In [25]:

```
!ls -l *.csv
```

```
-rw-r--r-- 1 jupyter jupyter 123363 Sep 18 12:16 taxi-test.csv
-rw-r--r-- 1 jupyter jupyter 578524 Sep 18 12:16 taxi-train.csv
-rw-r--r-- 1 jupyter jupyter 123872 Sep 18 12:16 taxi-valid.csv
```

We have 3 .csv files corresponding to train, valid, test. The ratio of file-sizes correspond to our split of the data.

In [26]:

```
%%bash
head taxi-train.csv
```

```
4.5,2010-04-29 12:28:00 UTC,-73.791475,40.851292,-73.942248,40.803482,2,0
33.7,2009-09-25 03:47:00 UTC,-73.992973,40.752672,-73.956852,40.610602,3,1
15.3,2009-04-02 14:08:58 UTC,-74.005741,40.72031,-73.967756,40.763629,1,2
11.0,2015-03-29 00:26:47 UTC,-73.97805786132812,40.7292366027832,-73.999015
80810547,40.72947692871094,2,3
5.7,2011-04-03 00:54:53 UTC,-73.968533,40.75412,-73.965823,40.757735,3,4
4.5,2012-02-21 11:53:00 UTC,-73.973462,40.748085,-73.982012,40.743467,1,5
6.9,2011-03-19 03:32:00 UTC,-74.004303,40.742632,-73.990107,40.767087,5,6
10.0,2013-03-07 08:19:51 UTC,-73.954258,40.778149,-73.96995,40.763363,1,7
9.0,2014-10-08 12:59:53 UTC,-73.990275,40.741188,-73.981148,40.750199,1,8
17.7,2012-03-04 00:57:00 UTC,-73.955437,40.764482,-73.87638,40.819622,6,9
```

Looks good! We now have our ML datasets and are ready to train ML models, validate them and evaluate them.

Benchmark

Before we start building complex ML models, it is a good idea to come up with a very simple model and use that as a benchmark.

My model is going to be to simply divide the mean fare_amount by the mean trip_distance to come up with a rate and use that to predict. Let's compute the RMSE of such a model.

In [27]:

```
def distance_between(lat1, lon1, lat2, lon2):
    # Haversine formula to compute distance "as the crow flies".
    lat1_r = np.radians(lat1)
    lat2_r = np.radians(lat2)
    lon_diff_r = np.radians(lon2 - lon1)
    sin_prod = np.sin(lat1_r) * np.sin(lat2_r)
    cos_prod = np.cos(lat1_r) * np.cos(lat2_r) * np.cos(lon_diff_r)
    minimum = np.minimum(1, sin_prod + cos_prod)
    dist = np.degrees(np.arccos(minimum)) * 60 * 1.515 * 1.609344

    return dist

def estimate_distance(df):
    return distance_between(
        df["pickuplat"], df["pickuplon"], df["dropofflat"], df["dropofflon

def compute_rmse(actual, predicted):
    return np.sqrt(np.mean((actual - predicted) ** 2))

def print_rmse(df, rate, name):
    print ("{1} RMSE = {0}".format(
        compute_rmse(df["fare_amount"], rate * estimate_distance(df)), nam

# TODO 4
FEATURES = ["pickuplon", "pickuplat", "dropofflon", "dropofflat", "passeng
TARGET = "fare_amount"
columns = list([TARGET])
columns.append("pickup_datetime")
columns.extend(FEATURES)  # in CSV, target is first column, after the feat
columns.append("key")
df_train = pd.read_csv("taxi-train.csv", header=None, names=columns)
df_valid = pd.read_csv("taxi-valid.csv", header=None, names=columns)
df_test = pd.read_csv("taxi-test.csv", header=None, names=columns)
rate = df_train["fare_amount"].mean() / estimate_distance(df_train).mean()
print ("Rate = ${0}/km".format(rate))
print_rmse(df_train, rate, "Train")
print_rmse(df_valid, rate, "Valid")
print_rmse(df_test, rate, "Test")
```

```
Rate = $2.6092296323555044/km
Train RMSE = 6.874900967905842
Valid RMSE = 9.255538325662519
Test RMSE = 9.085118516160394
```

Benchmark on same dataset

The RMSE depends on the dataset, and for comparison, we have to evaluate on the same dataset each time. We'll use this query in later labs:

In [28]:

```
validation_query = """
SELECT
    (tolls_amount + fare_amount) AS fare_amount,
    pickup_datetime,
    pickup_longitude AS pickuplon,
    pickup_latitude AS pickuplat,
    dropoff_longitude AS dropofflon,
    dropoff_latitude AS dropofflat,
    passenger_count*1.0 AS passengers,
    "unused" AS key
FROM
    `nyc-tlc.yellow.trips`
WHERE
    ABS(MOD(FARM_FINGERPRINT(CAST(pickup_datetime AS STRING)), 10000)) = 2
    AND trip_distance > 0
    AND fare_amount >= 2.5
    AND pickup_longitude > -78
    AND pickup_longitude < -70
    AND dropoff_longitude > -78
    AND dropoff_longitude < -70
    AND pickup_latitude > 37
    AND pickup_latitude < 45
    AND dropoff_latitude > 37
    AND dropoff_latitude < 45
    AND passenger_count > 0
"""

client = bigquery.Client()
df_valid = client.query(validation_query).to_dataframe()
print_rmse(df_valid, 2.59988, "Final Validation Set")
```

```
Final Validation Set RMSE = 8.135336354025382
```

The simple distance-based rule gives us a RMSE of **$8.14**. We have to beat this, of course, but you will find that simple rules of thumb like this can be surprisingly difficult to beat.

Let's be ambitious, though, and make our goal to build ML models that have a RMSE of less than $6 on the test set.