Introduction: Taxi Fare Prediction

Welcome to another Kaggle challenge. In this contest, the aim is to predict the fare of a taxi ride giver
latitude / longitude, and the number of passengers. This is a **supervised regression** machine learning

In this notebook, I'll provide you with a solid foundation and leave you with the challenge to better the
is an approachable problem and as usual with Kaggle competitions, provides realistic practice for bui
best way to learn is by doing, so let's work through a complete machine learning problem!

Great resources for Kaggle competitions are the [discussion forums] and the [kernels] completed by oth
adapting, and building on others' code, especially when you are getting started.

```
# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# Pandas display options
pd.set_option('display.float_format', lambda x: '%.3f' % x)

# Set random seed
RSEED = 100

# Visualizations
import matplotlib.pyplot as plt
%matplotlib inline

plt.style.use('fivethirtyeight')
plt.rcParams['font.size'] = 18

import seaborn as sns
palette = sns.color_palette('Paired', 10)
```

Read in 2 million rows and examine data

Throughout this notebook, we will work with only 2 million rows (out of 55 million). The first point for i
more data!

- **Potential improvement 1: use more data**

Generally, performance of a machine learning model increases as the amount of training data increas
returns, and I sample the data here in order to train faster. The data file is randomly sorted by date, so
sample in terms of time.

When we read in the data, we tell pandas to treat the `pickup_datetime` as a date. We will also drop th
does not tell us anything about the taxi trip. After reading in the data we'll remove any rows with `nan (`

```
data = pd.read_csv('../input/train.csv', nrows = 2_000_000,
                   parse_dates = ['pickup_datetime']).drop(columns = 'key')

# Remove na
data = data.dropna()
data.head()
```

	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	d
0	4.500	2009-06-15 17:26:21	-73.844	40.721	-73.842	
1	16.900	2010-01-05 16:52:16	-74.016	40.711	-73.979	
2	5.700	2011-08-18 00:35:00	-73.983	40.761	-73.991	
		2012-04-21				

Describe Data

An effective method for catching outliers and anomalies is to find the summary statistics for the data concentrate on the maxes and the minimums for finding outliers.

```
data.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitu
count	1999986.000	1999986.000	1999986.000	1999986.000	1999986.
mean	11.348	-72.523	39.930	-72.524	39.9
std	9.853	12.868	7.983	12.775	10.3
min	-62.000	-3377.681	-3458.665	-3383.297	-3461.5
25%	6.000	-73.992	40.735	-73.991	40.7
50%	8.500	-73.982	40.753	-73.980	40.7
75%	12.500	-73.967	40.767	-73.964	40.7
max	1273.310	2856.442	2621.628	3414.307	3345.9

Right away we can see there are a number of outliers in the `latitude` and `longitude` columns as we `passenger_count`. The target variable, `fare_amount` seems to have both negative values (unexpected unexpected).

Data Exploration and Data Cleaning

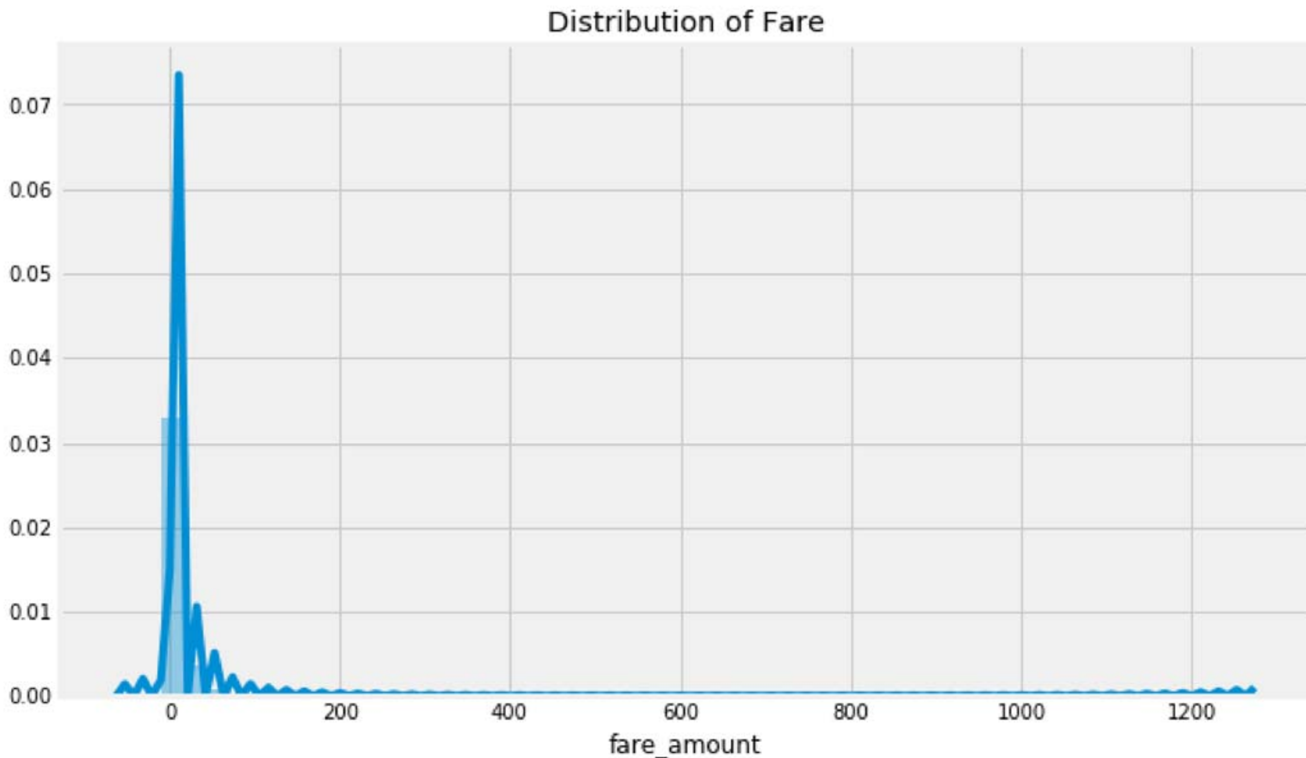
I often do data exploration and cleaning simulataneously. As I explore the data and find outlying value
to follow up later. Data cleaning usually involves domain knowledge (if applicable), statistical descript
similar problems. A good place for learning about a problem is the Kaggle discussions boards and otl

Examine the Target Variable

For a first graphical exploration, we can look at the distribution of the `fare_amount`, the target variable
use seaborn's `distplot` which shows both a kernel density estimate plot and a histogram.

```
plt.figure(figsize = (10, 6))
sns.distplot(data['fare_amount']);
plt.title('Distribution of Fare');
```

```
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using a
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```




```
print(f"There are {len(data[data['fare_amount'] < 0])} negative fares.")
print(f"There are {len(data[data['fare_amount'] == 0])} $0 fares.")
print(f"There are {len(data[data['fare_amount'] > 100])} fares greater than $100.")
```

```
There are 77 negative fares.
There are 56 $0 fares.
There are 785 fares greater than $100.
```

Remove Outliers

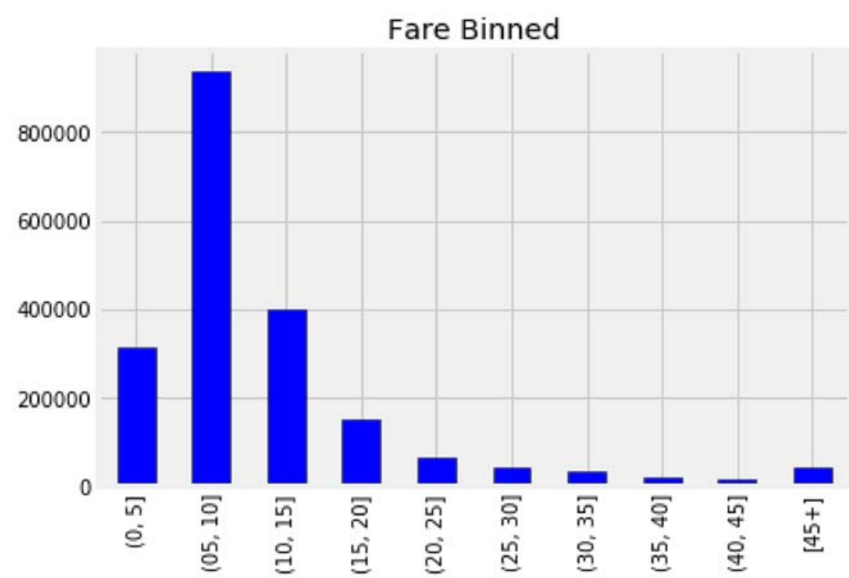
Based on this discussion on real-world taxi fares in New York City, I'm going to remove any fares less
minimum fare, so any values in the training set less than this amount must be errors in data collectior

Ill also remove any fares greater than $100. I'll justify this based on the limited number of fares outsid
that including these values helps the model! I'd encourage you to try different values and see which w

```
data = data[data['fare_amount'].between(left = 2.5, right = 100)]
```

For visualization purposes, I'll create a binned version of the fare. This divides the variable into a numl
into a discrete, categorical variable.

```
# Bin the fare and convert to string
data['fare-bin'] = pd.cut(data['fare_amount'], bins = list(range(0, 50, 5))).astype(str)

# Uppermost bin
data.loc[data['fare-bin'] == 'nan', 'fare-bin'] = '[45+]'

# Adjust bin so the sorting is correct
data.loc[data['fare-bin'] == '(5, 10]', 'fare-bin'] = '(05, 10]'

# Bar plot of value counts
data['fare-bin'].value_counts().sort_index().plot.bar(color = 'b', edgecolor = 'k');
plt.title('Fare Binned');
```



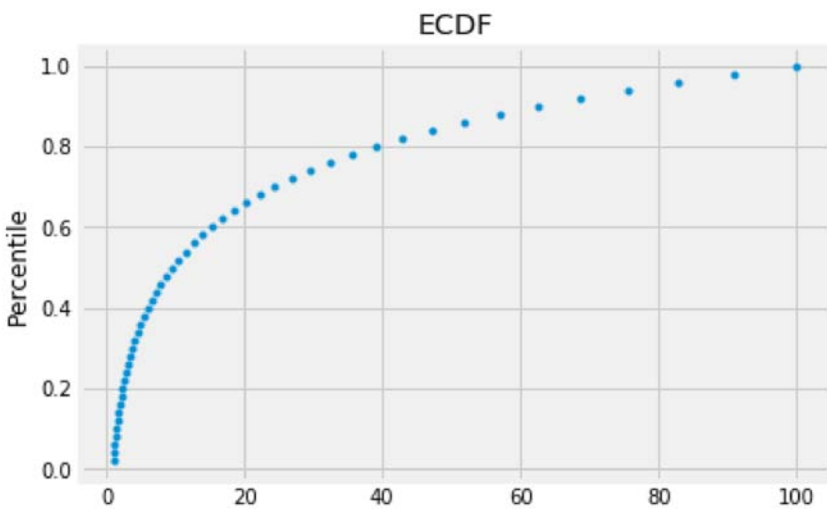
Empirical Cumulative Distribution Function Plot

Another plot for showing the distribution of a single variable is the empirical cumulative distribution fu
y-axis and the variable on the x-axis and gets around some of the issues associated with binning data
KDE.

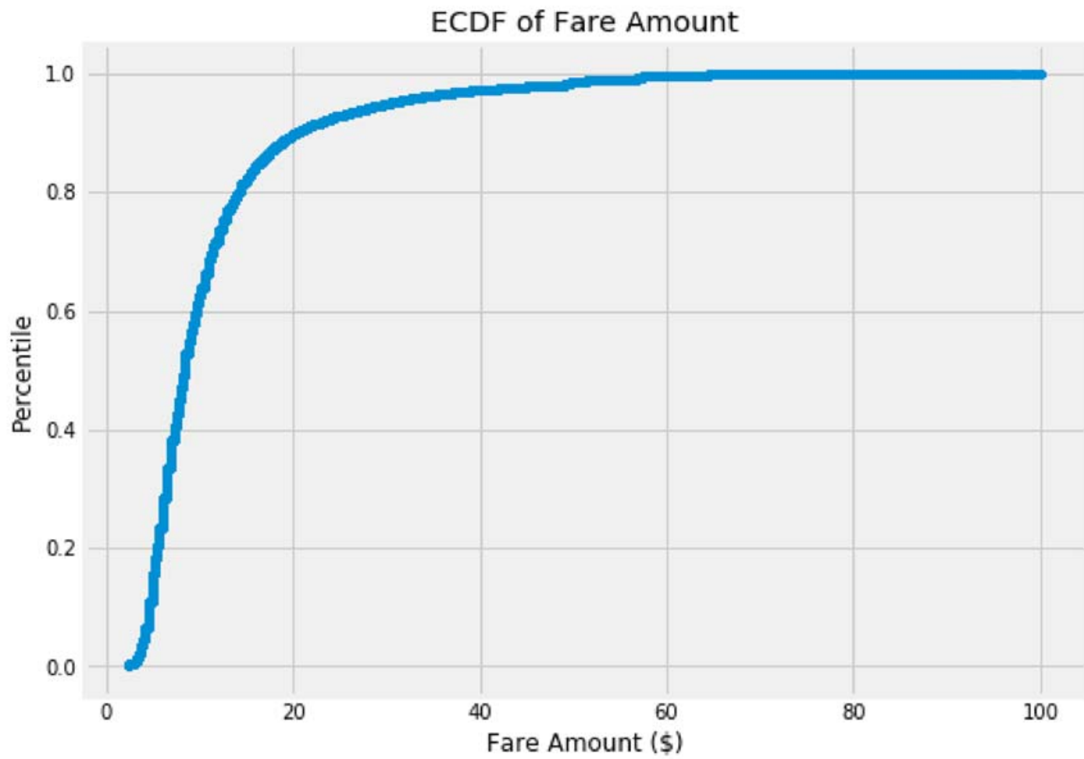
```
def ecdf(x):
    """Empirical cumulative distribution function of a variable"""
    # Sort in ascending order
    x = np.sort(x)
    n = len(x)

    # Go from 1/n to 1
    y = np.arange(1, n + 1, 1) / n

    return x, y
```

Below is an example of the ecdf. This plot is good for viewing outliers and also the percentiles of a di

```
xs, ys = ecdf(np.logspace(0, 2))
plt.plot(xs, ys, '.');
plt.ylabel('Percentile'); plt.title('ECDF');
```




```
xs, ys = ecdf(data['fare_amount'])
plt.figure(figsize = (8, 6))
plt.plot(xs, ys, '.')
plt.ylabel('Percentile'); plt.title('ECDF of Fare Amount'); plt.xlabel('Fare Amount ($)');
```



This shows the distribution is heavily right skewed. Most of the fares are below $20, with a heavy righ

Other Outliers

We can also remove observations based on outliers in other columns. First we'll make a graph of the some suspicious values.

↳ 10 cells hidden

Rides on Map of NYC

For a more contextualized representation, we can plot the pickup and dropoff on top of a map of New from https://www.kaggle.com/breemen/nyc-taxi-fare-data-exploration by Kaggle user [breeman](). All cr rest of his kernel for more excellent work!

The map was extracted from OpenStreetMaps ([https://www.openstreetmap.org/export#map=12/40.]()

↳ 10 cells hidden

Feature Engineering

Feature engineering is the process of creating features - predictor variables - out of a dataset. **Feature of the machine learning pipeline ([A Few Useful Things to Know about Machine Learning]()).** A model c and properly constructing features will determine how well your model performs.

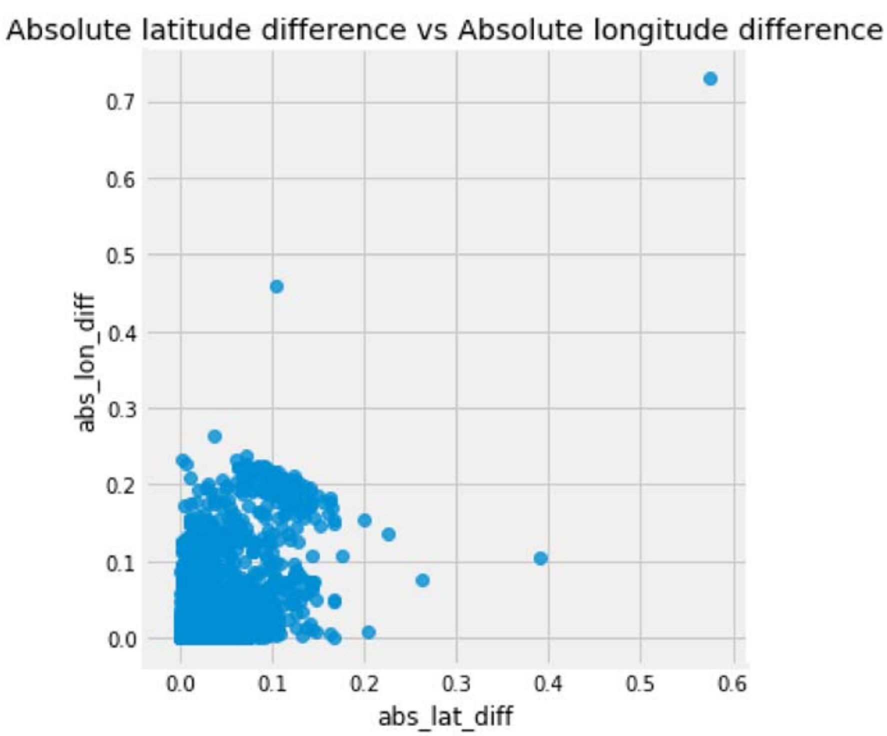
Feature engineering involves domain expertise and experience on prior machine learning projects. A g
competitions is other kernels. Feel free to use, adapt, and build on other's work!

Relative Distances in Latitude and Longitude

As a simple first step of feature engineering, we can find the absolute value of the difference in latituc `dropoff`. While this does not represent an actual distance (we would have to convert coordinate syst comparison of the distances of taxi rides. As the cost of a taxi ride is proportional to duration or dista distances to be a useful measure for estimating the fare.

```
# Absolute difference in latitude and longitude
data['abs_lat_diff'] = (data['dropoff_latitude'] - data['pickup_latitude']).abs()
data['abs_lon_diff'] = (data['dropoff_longitude'] - data['pickup_longitude']).abs()
```

```
sns.lmplot('abs_lat_diff', 'abs_lon_diff', fit_reg = False,
           data = data.sample(10000, random_state=RSEED));
plt.title('Absolute latitude difference vs Absolute longitude difference');
```



There do seem to be a few outliers, but I'll leave those in for now. We might also want to take a look if the absolute differences are 0.

```
no_diff = data[(data['abs_lat_diff'] == 0) & (data['abs_lon_diff'] == 0)]
no_diff.shape
```

It looks like there are 51,000 rides where the absolute latitude and longitude does not change! That se
worth following up!

Let's remake the plot above colored by the fare bin.

```
sns.lmplot('abs_lat_diff', 'abs_lon_diff', hue = 'fare-bin', size = 8, palette=palette,
           fit_reg = False, data = data.sample(10000, random_state=RSEED));
plt.title('Absolute latitude difference vs Absolute longitude difference');
```

```
sns.lmplot('abs_lat_diff', 'abs_lon_diff', hue = 'fare-bin', size = 8, palette = palette,
           fit_reg = False, data = data.sample(10000, random_state=RSEED));

plt.xlim((-0.01, .25)); plt.ylim((-0.01, .25))
plt.title('Absolute latitude difference vs Absolute longitude difference');
```

It does seem that the rides with a larger absolute difference in both longitude and latitude tend to cos
single variable, we can add up the two differences in latitude and longitude and also find the square r
The former feature would be called the Manhattan distance - or l1 norm - and the latter is called the E
these distances are specific examples of the general Minkowski distance.

Manhattan and Euclidean Distance

The Minkowski Distance between two points is expressed as:

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

if p = 1, then this is the Manhattan distance and if p = 2 this is the Euclidean distance. You may also s
where the number indicates p in the equation.

I should point out that these equations are only valid for actual distances in a cartesian coordinate sy
relative distances. To find the actual distances in terms of kilometers, we have to work with the latitu
system. This will be done later using the Haversine formula.

↳ 18 cells hidden

Read in test data and create same features

Before we forget, we need to read in the test data and create the same features. The test data must h
training data used in the model.

We can't exclude any of the test data based on outliers, and we also shouldn't use the test data for filt
data should ideally only be used a single time, to test the performance of a trained model.

For the test data, we need to save the `key` column for making submissions.

```
test = pd.read_csv('../input/test.csv', parse_dates = ['pickup_datetime'])

# Create absolute differences
test['abs_lat_diff'] = (test['dropoff_latitude'] - test['pickup_latitude']).abs()
test['abs_lon_diff'] = (test['dropoff_longitude'] - test['pickup_longitude']).abs()

# Save the id for submission
test_id = list(test.pop('key'))

test.describe()
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger
count	9914.000	9914.000	9914.000	9914.000	9
mean	-73.975	40.751	-73.974	40.752	
std	0.043	0.034	0.039	0.035	
min	-74.252	40.573	-74.263	40.569	
25%	-73.993	40.736	-73.991	40.735	
50%	-73.982	40.753	-73.980	40.754	
75%	-73.968	40.767	-73.964	40.769	
max	-72.987	41.710	-72.991	41.697	

No fare information here! It's our job to predict the fare for each test ride.

```
test['manhattan'] = minkowski_distance(test['pickup_longitude'], test['dropoff_longitude'],
                                       test['pickup_latitude'], test['dropoff_latitude'], 1)

test['euclidean'] = minkowski_distance(test['pickup_longitude'], test['dropoff_longitude'],
                                       test['pickup_latitude'], test['dropoff_latitude'], 2)
```

Calculate Distance between points using Haversine distance

To calculate a more realistic distance between the pickup and dropoff, we can use the [Haversine dist
representing the shortest distance along the surface of the Earth connecting two points taking into ac
(or so I'm told). It's not the best measure because the taxis do not travel along lines, but it's more accu

than the Manhattan and Euclidean distances made from the absolute latitude and longitude differenc distances are relative and do not take into account the spherical shape of the Earth.

(We could convert the latitude and longitude into cartesian coordinates after establishing an origin. O the Earth and another would be to use the average of all coordinates in the data as an origin. Then, or coordinate system, we could use the Manhattan and Euclidean formulas to find distances between pc approximations because we can't find the actual street distance.)

The formula for Haversine distance is:

$$= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2}{}\right.}\right.$$

where r is the radius of the Earth. The end units will be in km. My thanks go to this Stack Overflow ans https://stackoverflow.com/a/29546836

```
# Radius of the earth in kilometers
R = 6378

def haversine_np(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)

    All args must be of equal length.

    source: https://stackoverflow.com/a/29546836

    """
    # Convert latitude and longitude to radians
    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])

    # Find the differences
    dlon = lon2 - lon1
    dlat = lat2 - lat1

    # Apply the formula
    a = np.sin(dlat/2.0)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2.0)**2
    # Calculate the angle (in radians)
    c = 2 * np.arcsin(np.sqrt(a))
    # Convert to kilometers
    km = R * c

    return km
```

```
data['haversine'] =  haversine_np(data['pickup_longitude'], data['pickup_latitude'],
                         data['dropoff_longitude'], data['dropoff_latitude'])
```

```
test['haversine'] = haversine_np(test['pickup_longitude'], test['pickup_latitude'],
                         test['dropoff_longitude'], test['dropoff_latitude'])
```

```
subset = data.sample(100000, random_state=RSEED)

plt.figure(figsize = (10, 6))

for f, grouped in subset.groupby('fare-bin'):
    sns.kdeplot(grouped['haversine'], label = f'{f}', color = list(grouped['color'])[0]);

plt.title('Distribution of Haversine Distance by Fare Bin');
```

It does seem there is a significant difference here! The larger haversine distances tend to have larger to be close to those returned from the Euclidean and Manhattan calculations.

```
data.groupby('fare-bin')['haversine'].agg(['mean', 'count'])
```

```
data.groupby('fare-bin')['haversine'].mean().sort_index().plot.bar(color = 'g');
plt.title('Average Haversine Distance by Fare Amount');
plt.ylabel('Mean Haversine Distance');
```

```
sns.kdeplot(test['haversine']);
```

The test distribution seems to be similar to the training distribution.

As a final step, we can find the correlations between distances and fares.

```
corrs = data.corr()
corrs['fare_amount'].plot.bar(color = 'b');
plt.title('Correlation with Fare Amount');
```

All the measures of distance have a *positive* linear correlation with the fare, indicating that as they inc

The correlation coefficient measures the strength and direction of a linear relationship. Because the li is so strong, we may be able to just use a linear model (regression) to accurately predict the fares.

Machine Learning

Now that we have built a few potentially useful features, we can use them for machine learning: traini
the features. We'll start off with a basic model - Linear Regression - only using a few features and the
more features. There is reason to believe that for this problem, even a simple linear model will perfor
correlation of the distances with the fare. We generally want to use the simplest - and hence most int
accuracy threshold (dependent on the application) so if a linear model does the job, there's no need t
It's a best practice to start out with a simple model for just this reason!

First Model: Linear Regression

The first model we'll make is a simple linear regression using 3 features: the `abs_lat_diff`, `abs_lon_`
meant to serve as a baseline for us to beat.

It's good to start with a simple model because it will give you a baseline. Also, if a simple model work

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

lr = LinearRegression()
```

Create Training and Validation Set

We'll want to create a training and separate validation set to assess our model. Ideally, we only use th
model. We can make a validation set with 1 million observations to estimate our performance.

We **stratify** the split using the `fare-bin`. This ensures that the training and validation set have the sa
important for imbalanced classification problems, but it can also be useful for regression problems s
terms of the target in either the validation or training set. (We have to stratify based on a discrete vari

```
# Split data
X_train, X_valid, y_train, y_valid = train_test_split(data, np.array(data['fare_amount']),
                                                      stratify = data['fare-bin'],
                                                      random_state = RSEED, test_size = 1_000
```

Train with Simple Features

We'll train the linear regression using three features. The benefit of the linear regression is that it's int
and intercept.

```
lr.fit(X_train[['abs_lat_diff', 'abs_lon_diff', 'passenger_count']], y_train)

print('Intercept', round(lr.intercept_, 4))
print('abs_lat_diff coef: ', round(lr.coef_[0], 4),
```

```
      '\tabs_lon_diff coef:', round(lr.coef_[1], 4),
      '\tpassenger_count coef:', round(lr.coef_[2], 4))
```

```
Intercept 5.0179
abs_lat_diff coef:  115.7926    abs_lon_diff coef: 164.9234     passenger_count coef: 0.
```

In all cases, the coefficient is positive, indicating a larger value of the variable corresponds to a larger that according to a linear model, for every 1 more passenger, the fare increases by $0.02. The `interc` predicted if there is no latitude or longitude difference and the passenger count is 0.

Score Model

Here we use the validation set for assessing the model. We'll use two metrics:

- **Root mean squared error**: the metric used by the competition
- **Mean absolute percentage error**: the average percentage error of the predictions

I like using the mean absolute percentage error (MAPE) because it's often more interpretable.

```
from sklearn.metrics import mean_squared_error
import warnings
warnings.filterwarnings('ignore', category = RuntimeWarning)

def metrics(train_pred, valid_pred, y_train, y_valid):
    """Calculate metrics:
       Root mean squared error and mean absolute percentage error"""

    # Root mean squared error
    train_rmse = np.sqrt(mean_squared_error(y_train, train_pred))
    valid_rmse = np.sqrt(mean_squared_error(y_valid, valid_pred))

    # Calculate absolute percentage error
    train_ape = abs((y_train - train_pred) / y_train)
    valid_ape = abs((y_valid - valid_pred) / y_valid)

    # Account for y values of 0
    train_ape[train_ape == np.inf] = 0
    train_ape[train_ape == -np.inf] = 0
    valid_ape[valid_ape == np.inf] = 0
    valid_ape[valid_ape == -np.inf] = 0

    train_mape = 100 * np.mean(train_ape)
    valid_mape = 100 * np.mean(valid_ape)

    return train_rmse, valid_rmse, train_mape, valid_mape

def evaluate(model, features, X_train, X_valid, y_train, y_valid):
    """Mean absolute percentage error"""

    # Make predictions
```

```
    train_pred = model.predict(X_train[features])
    valid_pred = model.predict(X_valid[features])

    # Get metrics
    train_rmse, valid_rmse, train_mape, valid_mape = metrics(train_pred, valid_pred,
                                                              y_train, y_valid)

    print(f'Training:   rmse = {round(train_rmse, 2)} \t mape = {round(train_mape, 2)}')
    print(f'Validation: rmse = {round(valid_rmse, 2)} \t mape = {round(valid_mape, 2)}')
```

```
evaluate(lr, ['abs_lat_diff', 'abs_lon_diff', 'passenger_count'],
        X_train, X_valid, y_train, y_valid)
```

```
Training:   rmse = 5.33          mape = 27.92
Validation: rmse = 5.42          mape = 28.01
```

Without anything to compare these results to, we can't say if they are good. This is the reason for esta
machine learning!

Naive Baseline

To make sure that machine learning is even applicable to the task, we should compare these predictic
task, this can be as simple as the average value of the target in the training data.

```
train_mean = y_train.mean()

# Create list of the same prediction for every observation
train_preds = [train_mean for _ in range(len(y_train))]
valid_preds = [train_mean for _ in range(len(y_valid))]

tr, vr, tm, vm = metrics(train_preds, valid_preds, y_train, y_valid)

print(f'Baseline Training:   rmse = {round(tr, 2)} \t mape = {round(tm, 2)}')
print(f'Baseline Validation: rmse = {round(vr, 2)} \t mape = {round(vm, 2)}')
```

```
Baseline Training:   rmse = 9.37         mape = 64.89
Baseline Validation: rmse = 9.35         mape = 64.87
```

According to the naive baseline, our machine learning solution is effective! We are able to reduce the
generate much better predictions than using no machine learning. This should give us confidence we

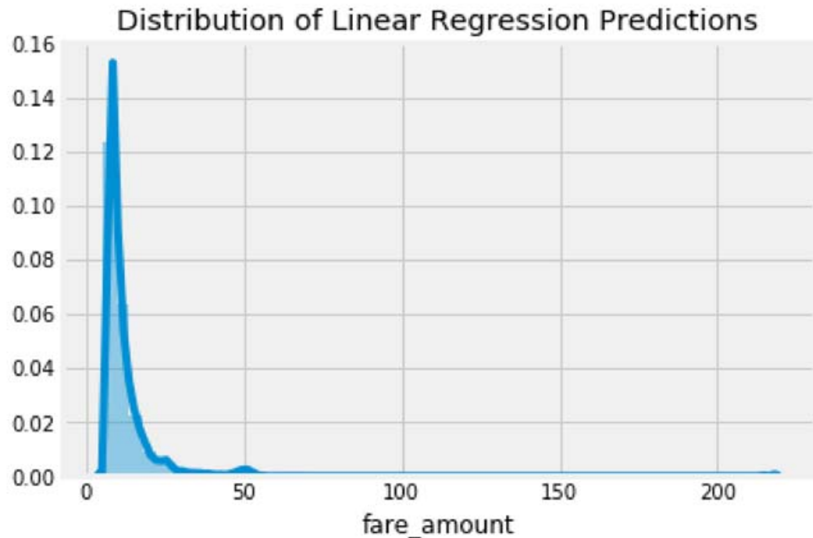
```
preds = lr.predict(test[['abs_lat_diff', 'abs_lon_diff', 'passenger_count']])
```

As a sanity check, we can plot the predictions.

```
sns.distplot(sub['fare_amount'])
```

```
plt.title('Distribution of Linear Regression Predictions');
```

```
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using a
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



The predicted distribution appears reasonable. Because the competition uses root mean squared erro far off will have an outsized effect on the error. Let's look at predictions that were greater than $100.

These three predictions that are all over $100 don't appear to be completely unexpected given the larg have to take a look at these predictions for other models and see if they agree.

As a linear model, the linear regression is not flexible at all. In other words, it has a high bias because the predictor variables (features) and the response (target). The final formula produced by the model right! In machine learning, we often have to make a tradeoff between model interpretability and mode regression does well, but as we'll see, a more complex model does even better.

Use More Features

While the first model scored well relative to the baseline, there is much room for improvement. As a fi we created, the `haversine` distance.

```
lr.fit(X_train[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count']], y_train)

evaluate(lr, ['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count'],
         X_train, X_valid, y_train, y_valid)
```

```
Training:   rmse = 5.0   mape = 24.4
Validation: rmse = 5.1   mape = 24.49
```

```
print(lr.intercept_)
```

```
print(lr.coef_)
```

Using this one more feature improved our score slightly. Here's another chance for improvement usin

- **Potential Improvement 3: find an optimal set of features or construct more features**. This can i combinations of features and evaluating them on the validation data. You can build additional fe researching the problem.

Collinear Features

One thing we do want to be careful about is highly correlated, known as collinear, features. These can of the model and lead to less interpretable models. Many of our features are already highly correlated plots the Pearson Correlation Coefficient for each pair of variables.

```
corrs = data.corr()

plt.figure(figsize = (12, 12))
sns.heatmap(corrs, annot = True, vmin = -1, vmax = 1, fmt = '.3f', cmap=plt.cm.PiYG_r);
```

You might not want to use two variables that are very highly correlated with each other (such as `eucl`
with interpretability and performance.

Upgraded Model

When we want to improve performance, we generally have a few options:

1. Get more data - either more observations or more variables
2. Engineer more / better features
3. Perform feature selection to remove irrelevant features
4. Try a more complex model
5. Perform hyperparameter tuning of the selected model

We already saw that including another feature could improve perfomance. For now let's move past th come back to features later).

The simple linear regression has no hyperparameters to optimize (no settings to tune) so we'll try app well, we can use it for testing additional features or performing feature selection

Non-Linear Model: Random Forest

For a first non-linear model, we'll use the [Random Forest](#) regressor. This is a powerful ensemble of re and generalization ability because of its low variance. We'll use most of the default hyperparameters `max_depth` of each tree in the forest. For the features, we'll use the four features which delivered goo

```
from sklearn.ensemble import RandomForestRegressor

# Create the random forest
random_forest = RandomForestRegressor(n_estimators = 20, max_depth = 20,
                                      max_features = None, oob_score = True,
                                      bootstrap = True, verbose = 1, n_jobs = -1)

# Train on data
random_forest.fit(X_train[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count']],
```

```
evaluate(random_forest, ['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count'],
         X_train, X_valid, y_train, y_valid)
```

The random forest does much better than the simple linear regression. This indicates that the proble not linear in terms of the features we have constructed. From here going forward, we'll use the same increased performance.

Overfitting

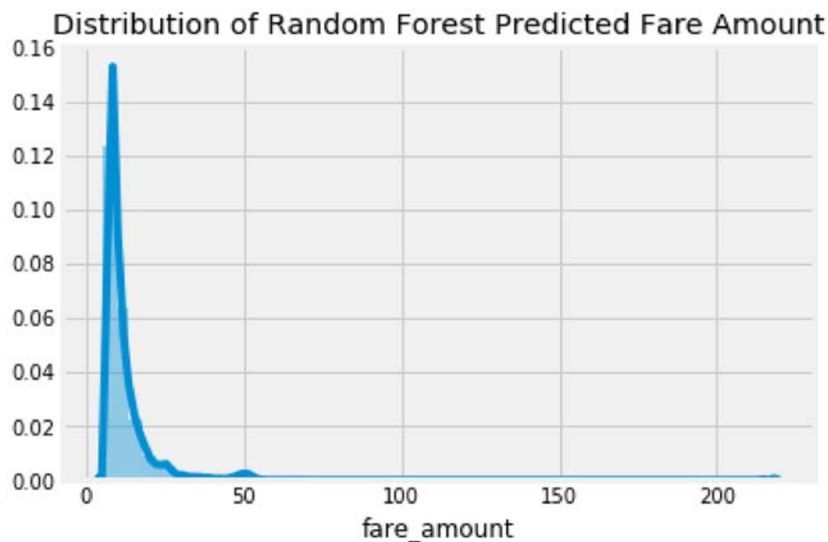
Given the gap between the training and the validation score, we can see that our model is **overfitting** most common problems in machine learning and is usually addressed either by training with more da the model. This leads to another recommendation for improvement:

- **Potential Improvement 4: Try searching for better random forest model hyperparameters**. You `RandomizedSearchCV` a useful tool.

I'll provide some starter code for hyperparameter optimization later in the notebook.

```
preds = random_forest.predict(test[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_c

sns.distplot(sub['fare_amount'])
plt.title('Distribution of Random Forest Predicted Fare Amount');
```

```
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  20 out of  20 | elapsed:    0.1s finished
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using a
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



This time we don't see any extreme predictions as we saw with the first linear regression. The random predictions because the voting of the trees means that any single tree that estimates an extreme valu predictions.

Let's look at the 3 predictions the original simple linear regression estimated as over $100.

Wow! The random forest and the linear regression significantly disagree. This brings up another point combine the predictions of multiple models. Oftentimes, averaging the predictions of multiple models either model by itself. Just for fun, let's try averaging the validation predictions of both models.

Average Models

We'll assess the validation performance of a simple averaging of the linear regression and random fo

```
lr_tpred = lr.predict(X_train[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count'
rf_tpred = random_forest.predict(X_train[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passe
```

```
lr_pred = lr.predict(X_valid[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passenger_count']]
rf_pred = random_forest.predict(X_valid[['haversine', 'abs_lat_diff', 'abs_lon_diff', 'passen

# Average predictions
train_pred = (lr_tpred + rf_tpred) / 2
valid_pred = (lr_pred + rf_pred) / 2

tr, vr, tm, vm = metrics(train_pred, valid_pred, y_train, y_valid)

print(f'Combined Training:   rmse = {round(tr, 2)} \t mape = {round(tm, 2)}')
print(f'Combined Validation: rmse = {round(vr, 2)} \t mape = {round(vm, 2)}')
```

```
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  20 out of  20 | elapsed:    4.4s finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
Combined Training:   rmse = 3.89         mape = 21.16
Combined Validation: rmse = 4.43         mape = 22.87
[Parallel(n_jobs=4)]: Done  20 out of  20 | elapsed:    4.8s finished
```

For this problem, the random forest by itself is slightly better. However, I'd encourage you to experime best model depends on the dataset.

More Features

Now that we've decided on the Random Forest as our model, we can try using additional features. Let the features for training. The function below trains the random forest and assesses it on the validatio

```
def model_rf(X_train, X_valid, y_train, y_valid, test, features,
             model = RandomForestRegressor(n_estimators = 20, max_depth = 20,
                                           n_jobs = -1),
             return_model = False):
    """Train and evaluate the random forest using the given set of features."""

    # Train
    model.fit(X_train[features], y_train)

    # Validation
    evaluate(model, features, X_train, X_valid, y_train, y_valid)

    # Make predictions on test and generate submission dataframe
    preds = model.predict(test[features])
    sub = pd.DataFrame({'key': test_id, 'fare_amount': preds})

    # Extract feature importances
    feature_importances = pd.DataFrame({'feature': features,
                                        'importance': model.feature_importances_}).\
                          sort_values('importance', ascending = False).set_index('feature')
```

```
    if return_model:
        return sub, feature_importances, model

    return sub, feature_importances
```

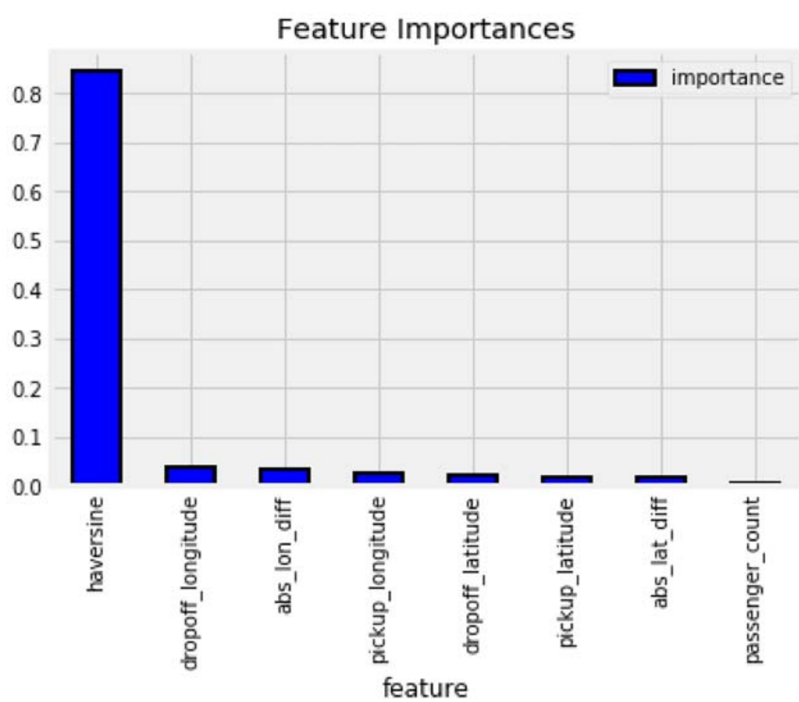
```
data.columns
```

```
# Evaluate using 8 features
sub, fi = model_rf(X_train, X_valid, y_train, y_valid, test,
                   features = ['abs_lat_diff', 'abs_lon_diff', 'haversine', 'passenger_count'
                               'pickup_latitude', 'pickup_longitude', 'dropoff_latitude', 'dr
```

It appears that using more features helps the model! We can look at the feature importances to see w

Feature Importances

```
fi.plot.bar(color = 'b', edgecolor = 'k', linewidth = 2);
plt.title('Feature Importances');
```



The `haversine` distance is by far the most important with the other features showing considerably le: that distance is key, and we might want to find a more accurate way of calculating distances.

Additional Feature Engineering

We saw that adding more features improves the performance of the model. A natural progression is t have not made any use of the `pickup_datetime` which provides the precise moment of pickup and th

Extract Datetime Information

We can write a simple function that extracts as much date and time information from a datetime as p excellent fast.ai library, in particular, the structured module (available at https://github.com/fastai/fas have made a few changes based on what's worked best for me in the past on time-series problems.

This function calculates a number of expected attributes:

- Year
- Month
- Day
- Day of Year
- Day of Week

When we have a time, additional variables can be calculated:

- Hour
- Minute
- Second

Pandas also offers options for some more complex attributes like:

- Is_month_end
- Is_month_start
- Is_quarter_end
- Is_quarter_start
- Is_year_end
- Is_year_start

These are more intended for financial analysis, but they may come in handy in other problems. For no attributes. For more information on datetimes in Pandas, refer to the documentation.

Fractional Time Variables

Finally, I add a number of other calculations that combine existing measures to create fractional varia

- Fractional time of day
- Fractional time of week
- Fractional time of month
- Fractional time of year

These are all measured from 0 - 1 in units of whichever time period we are measuring on. The idea be
the place of several other time indicators. For example, instead of using the `Dayof week`, `Hour`, `Minut`
time of the week to find out precisely when the observation takes place in the week.

I have found that the fractional time variables work well in practice especially with non-linear models.
approach is [cyclical variable encoding]() of time features, but I haven't found this to be necessary with n

```
import re

def extract_dateinfo(df, date_col, drop=True, time=False,
                     start_ref = pd.datetime(1900, 1, 1),
                     extra_attr = False):
    """
    Extract Date (and time) Information from a DataFrame
    Adapted from: https://github.com/fastai/fastai/blob/master/fastai/structured.py
    """
    df = df.copy()

    # Extract the field
    fld = df[date_col]

    # Check the time
    fld_dtype = fld.dtype
    if isinstance(fld_dtype, pd.core.dtypes.dtypes.DatetimeTZDtype):
        fld_dtype = np.datetime64

    # Convert to datetime if not already
    if not np.issubdtype(fld_dtype, np.datetime64):
        df[date_col] = fld = pd.to_datetime(fld, infer_datetime_format=True)


    # Prefix for new columns
    pre = re.sub('[Dd]ate', '', date_col)
    pre = re.sub('[Tt]ime', '', pre)

    # Basic attributes
    attr = ['Year', 'Month', 'Week', 'Day', 'Dayofweek', 'Dayofyear', 'Days_in_month', 'is_le

    # Additional attributes
    if extra_attr:
        attr = attr + ['Is_month_end', 'Is_month_start', 'Is_quarter_end',
                       'Is_quarter_start', 'Is_year_end', 'Is_year_start']

    # If time is specified, extract time information
    if time:
        attr = attr + ['Hour', 'Minute', 'Second']

    # Iterate through each attribute
    for n in attr:
        df[pre + n] = getattr(fld.dt, n.lower())
```

```
    # Calculate days in year
    df[pre + 'Days_in_year'] = df[pre + 'is_leap_year'] + 365

    if time:
        # Add fractional time of day (0 - 1) units of day
        df[pre + 'frac_day'] = ((df[pre + 'Hour']) + (df[pre + 'Minute'] / 60) + (df[pre + 'S

        # Add fractional time of week (0 - 1) units of week
        df[pre + 'frac_week'] = (df[pre + 'Dayofweek'] + df[pre + 'frac_day']) / 7

        # Add fractional time of month (0 - 1) units of month
        df[pre + 'frac_month'] = (df[pre + 'Day'] + (df[pre + 'frac_day'])) / (df[pre + 'Days_

        # Add fractional time of year (0 - 1) units of year
        df[pre + 'frac_year'] = (df[pre + 'Dayofyear'] + df[pre + 'frac_day']) / (df[pre + 'D

    # Add seconds since start of reference
    df[pre + 'Elapsed'] = (fld - start_ref).dt.total_seconds()

    if drop:
        df = df.drop(date_col, axis=1)

    return df
```

```
print(data['pickup_datetime'].min())
print(test['pickup_datetime'].min())
```

```
2009-01-01 00:00:46
2009-01-01 11:04:24
```

For a reference time, we can use the start of the training data. This means that the `Elapsed` measure observations.

```
test = extract_dateinfo(test, 'pickup_datetime', drop = False,
                         time = True, start_ref = data['pickup_datetime'].min())
test.head()
```

	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitu
0	2015-01-27 13:08:24	-73.973	40.764	-73.981	40.7
1	2015-01-27 13:08:24	-73.987	40.719	-73.999	40.7
2	2011-10-08 11:53:44	-73.983	40.751	-73.980	40.7
3	2012-12-01 21:12:12	-73.981	40.768	-73.990	40.7
4	2012-12-01 21:12:12	-73.966	40.790	-73.989	40.7

```
data = extract_dateinfo(data, 'pickup_datetime', drop = False,
                        time = True, start_ref = data['pickup_datetime'].min())
test.describe()
```

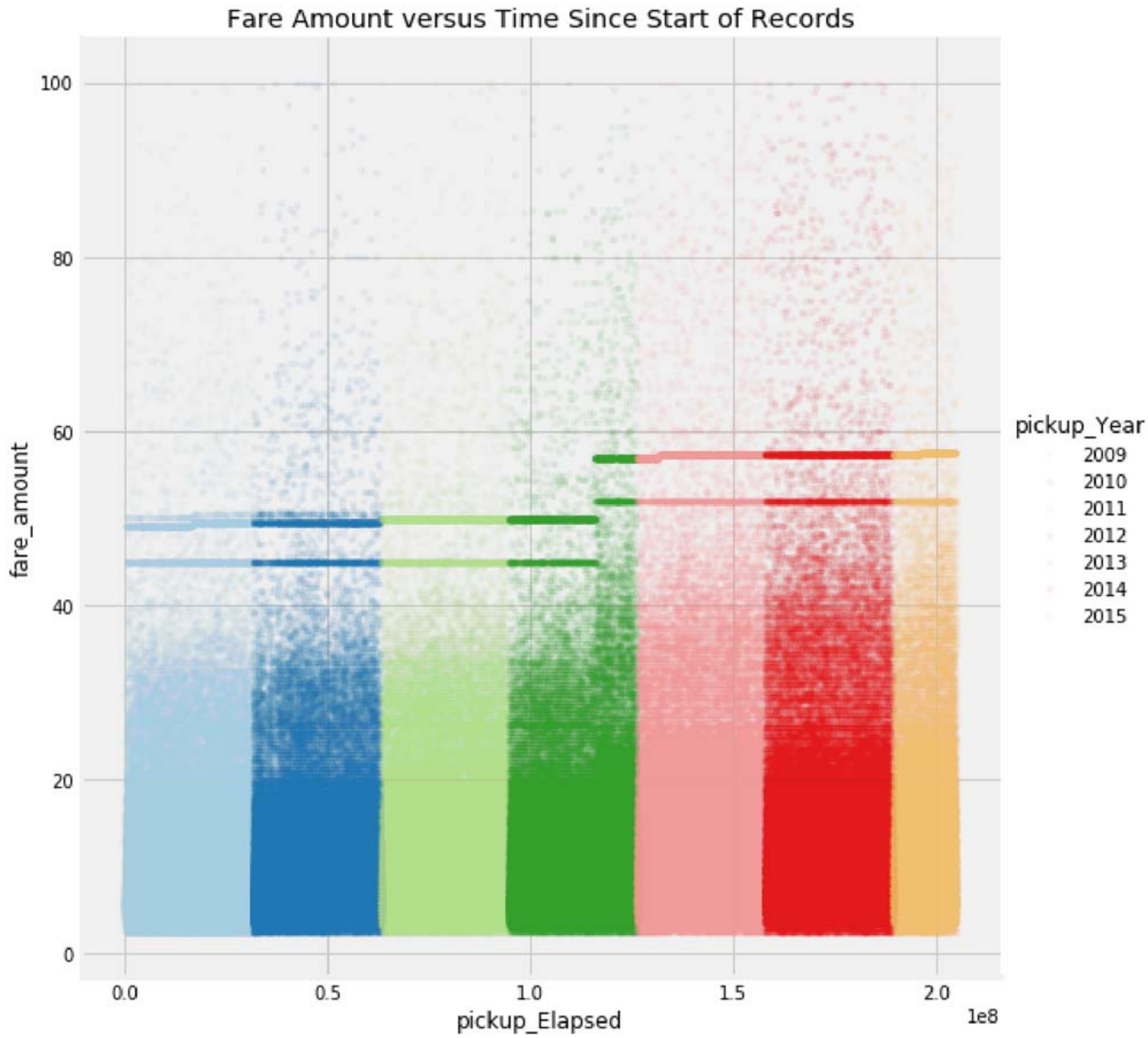
	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger
count	9914.000	9914.000	9914.000	9914.000	9
mean	-73.975	40.751	-73.974	40.752	
std	0.043	0.034	0.039	0.035	
min	-74.252	40.573	-74.263	40.569	
25%	-73.993	40.736	-73.991	40.735	
50%	-73.982	40.753	-73.980	40.754	
75%	-73.968	40.767	-73.964	40.769	
max	-72.987	41.710	-72.991	41.697	

Explore Time Variables

We now have a ton of time-variables to explore! First, let's ask the question if fares have increased ove `time_elapsed` versus the fare.

```
sns.lmplot('pickup_Elapsed', 'fare_amount', hue = 'pickup_Year', palette=palette, size = 8,
           scatter_kws= {'alpha': 0.05}, markers = '.', fit_reg = False,
           data = data.sample(1000000, random_state=RSEED));
plt.title('Fare Amount versus Time Since Start of Records');
```

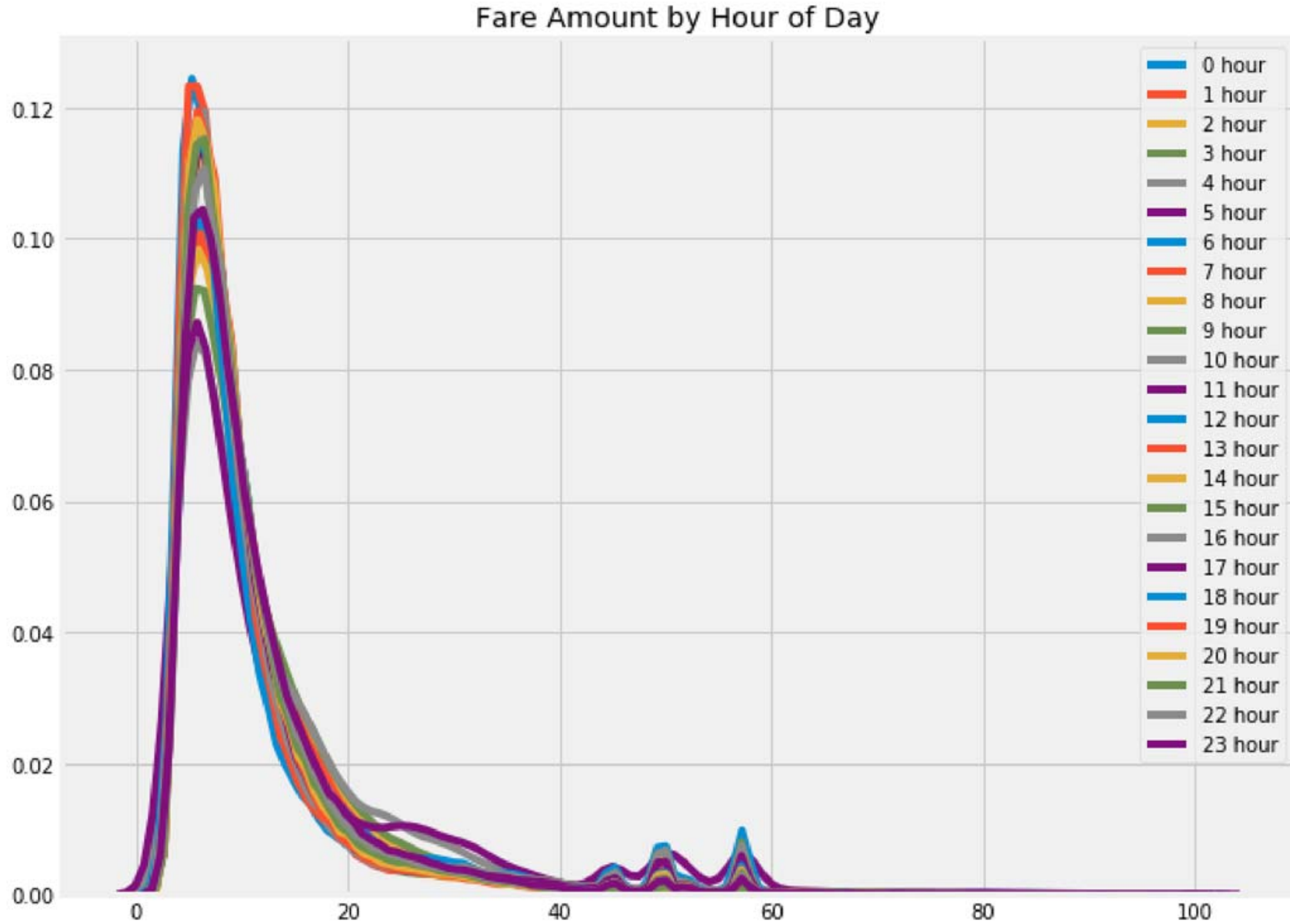
```
/opt/conda/lib/python3.6/site-packages/seaborn/regression.py:546: UserWarning: The `size
  warnings.warn(msg, UserWarning)
```



There appears to be a minor increase in prices over time which might be expected taking into accoun amount by the hour of day.

```
plt.figure(figsize = (10, 8))
for h, grouped in data.groupby('pickup_Hour'):
    sns.kdeplot(grouped['fare_amount'], label = f'{h} hour');
plt.title('Fare Amount by Hour of Day');
```

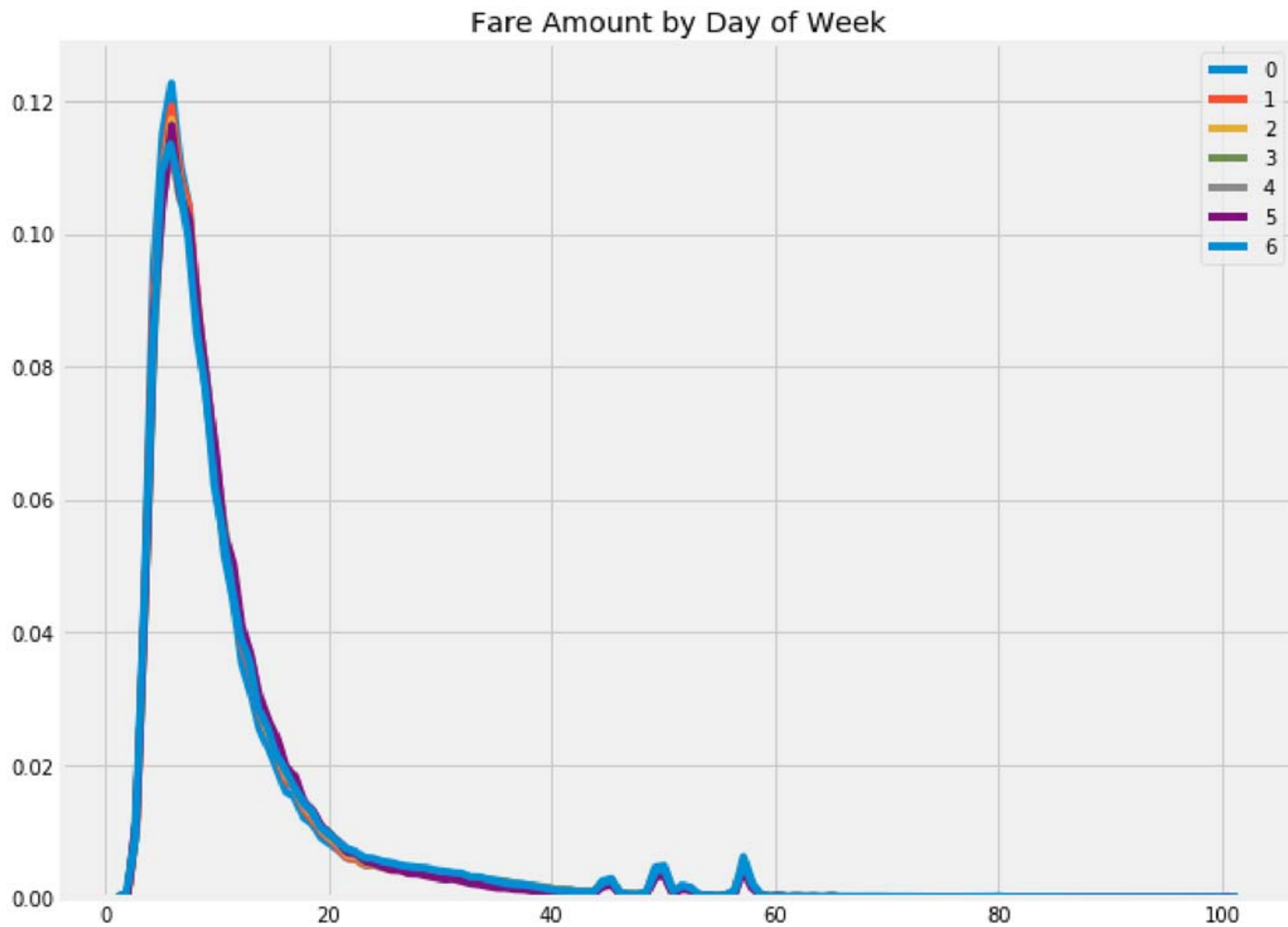
```
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using a
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



We can make the same plot by day of the week.

```
plt.figure(figsize = (10, 8))
for d, grouped in data.groupby('pickup_Dayofweek'):
    sns.kdeplot(grouped['fare_amount'], label = f'{d}')
plt.title('Fare Amount by Day of Week');
```

```
/opt/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using a
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

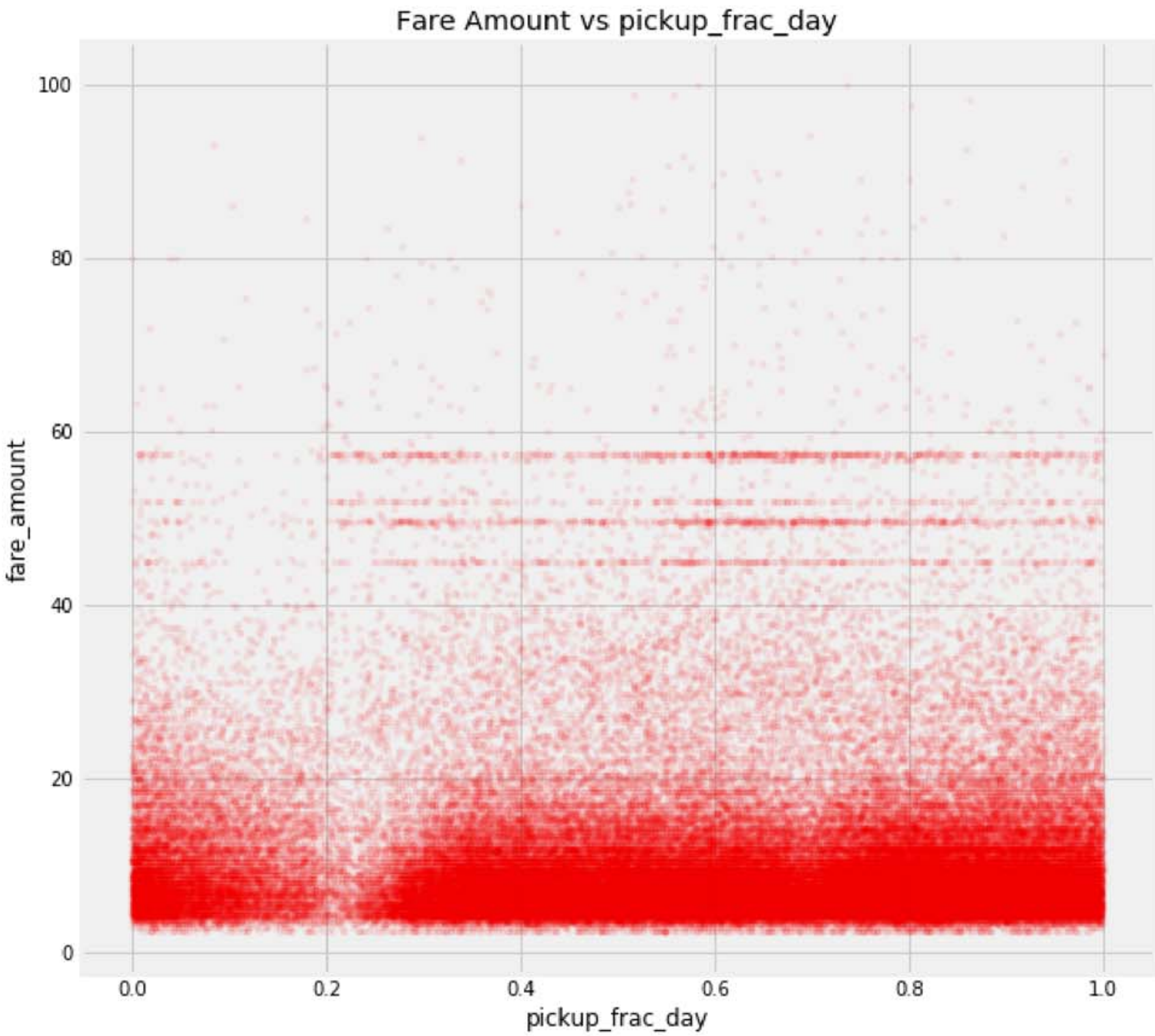


Both of these plots do not seem to show much difference between the different times.

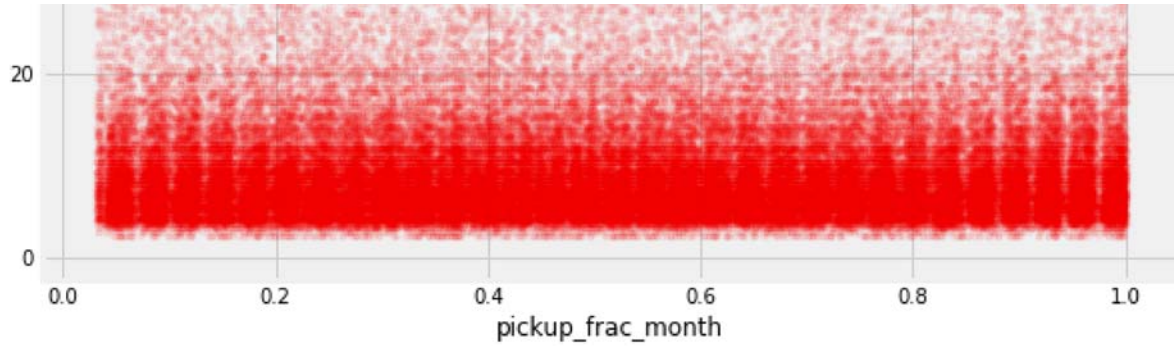
Fractional Time Plots

As a final exploration of the time variables, we can plot the fare amount versus the fractional time.

```
fig, axes = plt.subplots(2, 2, figsize = (20, 20))
axes = axes.flatten()

# Plot each of the fractional times
for i, d in enumerate(['day', 'week', 'month', 'year']):
    ax = axes[i]
    sns.regplot(f'pickup_frac_{d}', 'fare_amount',
                data = data.sample(100000, random_state = RSEED),
                fit_reg = False, scatter_kws = {'alpha': 0.05}, marker = '.', ax = ax,
                color = 'r')

    ax.set_title(f'Fare Amount vs pickup_frac_{d}')
```


Fare Amount vs pickup_frac_day
fare_amount
pickup_frac_day
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0


Fare Amount vs pickup_frac_month
fare_amount
40
60
80
100



None of these graphs are very decisive. One interesting thing to note is the horizontal bars at differen be certain routes that always have the same fare amount. We explored the fare distribution earlier, bu abnormalities in the fares.

```
fare_counts = data.groupby('fare_amount')['haversine'].agg(['count', pd.Series.nunique]).sort
fare_counts.head()
```

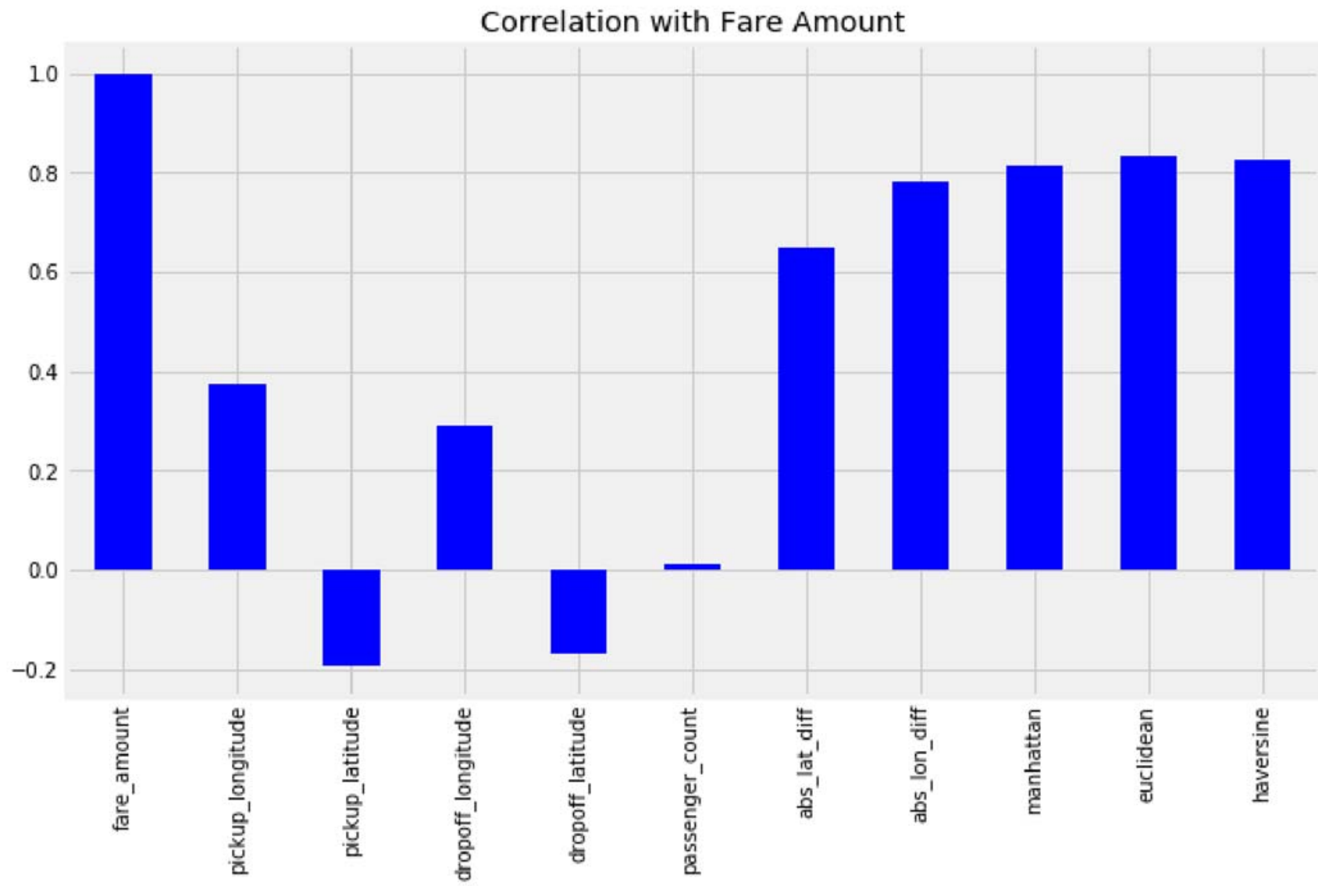
fare_amount	count	nunique
6.500	93466	92537.000
4.500	79044	77853.000
8.500	71764	71170.000
5.300	57042	56275.000
5.700	56692	55981.000

There are a number of very common fares. These could indicate certain rides that are a set amount. I standard fares, and if this information could be used for modeling. If there are set fares, the next step pickups and/or dropoffs.

Correlations with Target

Again we can show the correlations of all features with the target.

```
# corrs = data.corr()
corrs['fare_amount'].plot.bar(color = 'b', figsize = (10, 6));
plt.title('Correlation with Fare Amount');
```



It seems the most useful time variables may be the `Year` or `Elapsed` because most of the time featu
target. The `Elapsed` correlation is positive indicating that fares have tended to increase over time.

Test Time Features

Now we can use the time features in our model to see if they yield any improvement. We'll need to res
state).

```
X_train, X_valid, y_train, y_valid = train_test_split(data, np.array(data['fare_amount']),
                                                      stratify = data['fare-bin'],
                                                      random_state = RSEED, test_size = 1_000
```

For the time features, we'll use the fractional measurements for the day, week, and year, as well as the

```
time_features = ['pickup_frac_day', 'pickup_frac_week', 'pickup_frac_year', 'pickup_Elapsed']

features = ['abs_lat_diff', 'abs_lon_diff', 'haversine', 'passenger_count',
            'pickup_latitude', 'pickup_longitude',
            'dropoff_latitude', 'dropoff_longitude'] + time_features
```

```
# Test using the features
sub, fi = model_rf(X_train, X_valid, y_train, y_valid, test,
                   features = features)
```

The random forest does considerably better once we use the time features! As with the distance featu the new predictor variables we built are useful.

Just for comparison, we can go back to the linear regression and look at the performance.

```
lr = LinearRegression()

# Fit and evaluate
lr.fit(X_train[features], y_train)
evaluate(lr, features, X_train, X_valid, y_train, y_valid)
```

It seems that the new features helped both the random forest and the linear regression. Let's take a lo importances.

```
plt.figure(figsize = (10, 8))
fi['importance'].plot.bar(color = 'g', edgecolor = 'k');
plt.ylabel('Importance'); plt.title('Feature Importances');
```

Once again, the `haversine` distance dominates the importance. The time elapsed since the first recor although the other time features do not seem to be of much use.

Try with All Time Variables

For a final submission with the random forest, we'll use every single one of the features. This probably

```
features = list(data.columns)

for f in ['pickup_datetime', 'fare_amount', 'fare-bin', 'color']:
    features.remove(f)

len(features)
```

```
# Test using all the features
```

```
# Test using all the features
sub, fi, random_forest = model_rf(X_train, X_valid, y_train, y_valid, test,
                                  features = features, return_model = True)
```

```
plt.figure(figsize = (12, 7))
fi['importance'].plot.bar(color = 'g', edgecolor = 'k');
plt.ylabel('Importance'); plt.title('Feature Importances');
```

Visualize Validation Predicted Target

Using the trained random forest, we can make predictions on the validation set and plot the predictior potentially diagnose the model.

```
valid_preds = random_forest.predict(X_valid[features])

plt.figure(figsize = (10, 6))
sns.kdeplot(y_valid, label = 'Actual')
sns.kdeplot(valid_preds, label = 'Predicted')
plt.legend(prop = {'size': 30})
plt.title("Distribution of Validation Fares");
```

```
# Generate ecdf data
xv, yv = ecdf(valid_preds)
xtrue, ytrue = ecdf(y_valid)

# Plot the ecdfs on same plot
plt.scatter(xv, yv, s = 0.02,  c = 'r', marker = '.', label = 'Predicted')
plt.scatter(xtrue, ytrue, s = 0.02, c = 'b', marker = '.', label = 'True')
plt.title('ECDF of Predicted and Actual Validation')

plt.legend(markerscale = 100, prop = {'size': 20});
```

```
analyze = pd.DataFrame({'predicted': valid_preds, 'actual': y_valid})
analyze.describe()
```

At this point, our model is probably overfitting because we are using all the features, some of which a option for feature selection is to use only the most important features from the model.

Hyperparameter Tuning

With the random forest, there are a ton of model hyperparamters to optimize. The process of hyperpa best hyperparameters for an algorithm on a specific dataset. The ideal values changes across data se every new dataset. I like to think of hyperparameter optimization as finding the best settings for a ma

Random Search

We'll use a basic form of hyperparameter tuning, random search. This means constructing a paramet combinations of values, evaluating them in cross validation, and determining which combination perf the `RandomizedSearchCV` in Scikit-Learn.

The following code sets up the search. Feel free to play around with the `param_grid`.

```
from sklearn.model_selection import RandomizedSearchCV

# Hyperparameter grid
param_grid = {
    'n_estimators': np.linspace(10, 100).astype(int),
    'max_depth': [None] + list(np.linspace(5, 30).astype(int)),
    'max_features': ['auto', 'sqrt', None] + list(np.arange(0.5, 1, 0.1)),
    'max_leaf_nodes': [None] + list(np.linspace(10, 50, 500).astype(int)),
    'min_samples_split': [2, 5, 10],
    'bootstrap': [True, False]
}

# Estimator for use in random search
estimator = RandomForestRegressor(random_state = RSEED)

# Create the random search model
rs = RandomizedSearchCV(estimator, param_grid, n_jobs = -1,
                        scoring = 'neg_mean_absolute_error', cv = 3,
                        n_iter = 100, verbose = 1, random_state=RSEED)
```

We'll use a very limited sample of the data since random search is computationally expensive. Rando assess the model which means that for each combination of hyperparameters, we are training and te 3. This is another option that can be adjusted to determine if performance is affected.

```
tune_data = data.sample(100_000, random_state = RSEED)

# Select features
time_features = ['pickup_frac_day', 'pickup_frac_week', 'pickup_frac_year', 'pickup_Elapsed']

features = ['abs_lat_diff', 'abs_lon_diff', 'haversine', 'passenger_count',
            'pickup_latitude', 'pickup_longitude',
            'dropoff_latitude', 'dropoff_longitude'] + time_features

rs.fit(tune_data[features], np.array(tune_data['fare_amount']))
```

```
model = rs.best_estimator_
print(f'The best parameters were {rs.best_params_} with a negative mae of {rs.best_score_}')
```

Evaluate Best Model from Random Search

The best model from random search is available through the `best_estimator_` attribute of the fitted r
refits the best estimator on all the data we give it, but since this was only a sample of the full data, we
retraining.

↳ 3 cells hidden