Statistical Models to Predict Electric Vehicle Ownership

Fall, 2018

Kho, Jeremy Aaron

Abstract (5 points)

The transition to electric vehicles seems to be inevitable and predictions are being made about when total adoption will happen. The reality is that EV adoption will span years and will require changes to policies and infrastructure. One problem is the lack of data related to EV adoption at this scale. However, looking to states that have higher adoption serves as a good start. In this project, I explore statistical methods to predict EV ownership among households in the near future. Due to its high adoption rate and availability of data from the 2017 National Household Travel Survey, California is the basis of the model. Because only 2.5% of California households own electric vehicles, Precision-Recall through the F1 Score was selected as the evaluation metric. After data preparation, the records were split into sets for training, validation, and testing. Three models were constructed: k-Nearest Neighbors, Random Forest, and Ridge Regression. To provide meaningful predictions, the threshold for classification was set as a dynamic parameter tuned through the validation set. In testing, kNN performed the worst with an F Score at 0.53, followed by Random Forest at 0.55, and Ridge at 0.57. The best precision and recall scores for predicting EV owners were at 14% and 22%, respectively, which are seen to be modest, considering that only 2.3% of the test set were actually owners. Although model predictions are better than random, improvements clearly need to be made. A prediction made in Texas with a six-fold increase in adoption can be used by analyzing which categories of households were false positives and would be most similar to EV owners, making them 'low-hanging fruit' for decision makers. Aside from the meager performance of the models, caveats include factors not included, such as state culture, politics, and incentives. Moreover, the data are a snapshot in time and do not incorporate the rate of adoption, change in price of vehicles, change in price of fuel, change in environmental policy, and so on.

Project Background (5 points)

When you watch or read the news, the transition to electric vehicles seems to be inevitable - Tesla's stock continues to surge and all major automakers have shifted their focus and resources to developing and manufacturing electric vehicles. Predictions are being made about when total adoption will happen and the hype continues to grow. However, the reality is that this change won't happen overnight. The adoption of EVs is a gradual process that will span years and will require many changes to our policies and infrastructure.

Local and state governments are on the frontlines of enacting such policy changes, building charging stations, and upgrading electric grids. This can be very difficult to do. How can communities plan for a future that's yet to happen? How can planners determine the number of electric vehicles to expect in the coming years? How can engineers determine the geography of adoption that will inform infrastructure decisions?

On the other hand, while the intentions to facilitate the transition to electric vehicles is good, there is an important caveat to note. Electric vehicles, at least for now, are relatively expensive and are likely to be purchased by wealthier households. Similar to some policies related to residential solar panels, the diversion of resources to geographies most likely to adopt EVs may disproportionately benefit wealthier households. Although this is not the main focus of this project, governments need to be able to identify households that are less likely to adopt EVs in order to adjust policies to facilitate social equity in the transition that is to come.

One key problem is the lack of data related to electric vehicle adoption at this scale. For context, transportation surveys are usually conducted about once a decade and are very expensive. However, some states are more advanced in electric vehicle adoption than others. Although there are many differences between states that are difficult to quantify or model, such as culture and politics, looking to states that are further along the adoption curve would serve as a good springboard for analysis.

Project Question (5 points)

In this project, I explore the use of statistical methods to predict whether households are likely or unlikely to adopt electric vehicles in the near future.

By mapping households that are most likely to adopt EVs, local governments and utility companies can sufficiently plan for new or upgraded facilities to accommodate the change. Aside from just the number of EVs to expect, the characteristics of households can help determine the geographic locations where action is most appropriate. At the same time, the characteristics of households unlikely to adopt EVs may inform decision-makers to implement programs to incentivize adoption or enact policies to accommodate them in other ways.

By the end of the project, I will use the model to predict the number of potential EV-owning households in a particular state.

Input Data Description (5 points)

The primary dataset used in this project is the 2017 US National Household Travel Survey (https://nhts.ornl.gov/), which contains four sets of data - Households, Persons, Trips, and Vehicles. For this project, I utilized the data on households and vehicles. The data were collected through surveys conducted from April 2016 to May 2017 by the Federal Highway Administration (FHWA) with members of a sample size of households across the country. Documentation available for the dataset are the Data User Guide (https://nhts.ornl.gov/assets/2017UsersGuide.pdf) and the Codebook (https://nhts.ornl.gov/assets/codebook_v1.1.pdf).

In [518]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings

warnings.filterwarnings('ignore')
plt.style.use('fivethirtyeight')
```

First, we take a look at the household information.

Structure - The data was provided in a CSV format. Each record is a household that was part of the survey sample. Specifically, there was a primary respondent who answered for the entire household. Most of the columns are integer types, with a few object or string types, and a single float type for the household sample weight. Although most of the data are represented numerically, most are encoded as ordered categories, with bins of values representing a range of characteristics.

Granularity - All of the records are consistently individual households, but come from different states, locations, and demographics. Because it was conducted through a survey, it is simply a snapshot in time and has no temporal information.

Scope - The time range of surveys is between April 2016 and May 2017. Household surveyed come from different states, locations, and demographics. Features include household information, geographic characteristics, travel patterns, lifestyles, and opinions of respondents.

Temporality - All the surveys were conducted between April 2016 and May 2017. Hence, the data is a snapshot in time. While there are other surveys conducted in the past, such as the 2009 travel survey and the 2010 census, those datasets are not included in this study because they lack some pertinent features. Furthermore, because the data was collected through surveys, the information provided represent typical travel days not actual ones, from the perspective of the respondent.

Faithfulness - Most of the information were obtained through household surveys. Hence, they are prone to the mistakes and biases of the respondents. Some information, such as geographical characteristics, were obtained through other government records and classifications. Hence, those data are reliant on the methodologies imposed there.

In [2]:
```
# Quick View
hhpub = pd.read_csv('Csv/hhpub.csv')
hhpub.head()
```

Out[2]:

	HOUSEID	TRAVDAY	SAMPSTRAT	HOMEOWN	HHSIZE	HHVEHCNT	HHFAMINC	PC	SPHONE	TAB	...	SMPLSI
0	30000007	2	3	1	3	5	7	2	1	2	...	2
1	30000008	5	2	1	2	4	8	1	1	2	...	2
2	30000012	5	3	1	1	2	10	1	1	3	...	2
3	30000019	5	3	1	2	2	3	1	5	5	...	2
4	30000029	3	3	1	2	2	5	2	5	1	...	2

5 rows × 58 columns

In [3]:
```
# Column Names
hhpub.columns
```

Out[3]:
```
Index(['HOUSEID', 'TRAVDAY', 'SAMPSTRAT', 'HOMEOWN', 'HHSIZE', 'HHVEHCNT',
       'HHFAMINC', 'PC', 'SPHONE', 'TAB', 'WALK', 'BIKE', 'CAR', 'TAXI', 'BUS',
       'TRAIN', 'PARA', 'PRICE', 'PLACE', 'WALK2SAVE', 'BIKE2SAVE', 'PTRANS',
       'HHRELATD', 'DRVRCNT', 'CNTTDHH', 'HHSTATE', 'HHSTFIPS', 'NUMADLT',
       'YOUNGCHILD', 'WRKCOUNT', 'TDAYDATE', 'HHRESP', 'LIF_CYC', 'MSACAT',
       'MSASIZE', 'RAIL', 'URBAN', 'URBANSIZE', 'URBRUR', 'SCRESP', 'CENSUS_D',
       'CENSUS_R', 'CDIVMSAR', 'HH_RACE', 'HH_HISP', 'HH_CBSA', 'RESP_CNT',
       'WEBUSE17', 'SMPLSRCE', 'WTHHFIN', 'HBHUR', 'HTHTNRNT', 'HTPPOPDN',
       'HTRESDN', 'HTEEMPDN', 'HBHTNRNT', 'HBPPOPDN', 'HBRESDN'],
      dtype='object')
```

In [296]:
```
# Column Types
hhpub.dtypes.value_counts()
```

Out[296]:
```
int64      54
object      3
float64     1
dtype: int64
```

Next, we take a look at the vehicle information.

Structure - The data was provided in a CSV format. Each record is a vehicle owned by a household that is part of the survey sample. Specifically, there was a primary respondent who answered for the entire household. Similar to the previous dataset, most of the columns are integer types, with a few object or string types, and a couple of float types for the household sample weight and vehicle mileage. Although most of the data are represented numerically, most are encoded as ordered categories, with bins of values representing a range of characteristics.

Granularity - All of the records are consistently individual vehicles, but are owned by different households, and come from different states, locations, and demographics. Because it was conducted through a survey, it is simply a snapshot in time and has no temporal information.

Scope - The time range of surveys is between April 2016 and May 2017. Households surveyed come from different states, locations, and demographics. Features include repeated information from the households database, but include vehicle specifications and characteristics.

Temporality - All the surveys were conducted between April 2016 and May 2017. Hence, the data is a snapshot in time. While there are other surveys conducted in the past, such as the travel surveys in 2009 and decades prior, those datasets are not included in this study because they lack some pertinent features.

Faithfulness - Most of the information were obtained through household surveys. Hence, they are prone to the mistakes and biases of the respondents. Some information, such as geographical characteristics of the households that own the vehicles, were obtained through other government records and classifications. Hence, those data are reliant on the methodologies imposed there.

In [8]:
```
# Quick View
vehpub = pd.read_csv('Csv/vehpub.csv')
vehpub.head()
```

Out[8]:

	HOUSEID	VEHID	VEHYEAR	VEHAGE	MAKE	MODEL	FUELTYPE	VEHTYPE	WHOMAIN	OD_READ	...	BEST_EI
0	30000007	1	2007	10	49	49032	1	1	3	69000	...	-1
1	30000007	2	2004	13	49	49442	1	2	-8	164000	...	-1
2	30000007	3	1998	19	19	19014	1	1	1	120000	...	-1
3	30000007	4	1997	20	19	19021	1	1	2	-88	...	-1
4	30000007	5	1993	24	20	20481	1	4	2	300000	...	-1

5 rows × 54 columns

In [9]:
```
# Column Names
vehpub.columns
```

Out[9]:
```
Index(['HOUSEID', 'VEHID', 'VEHYEAR', 'VEHAGE', 'MAKE', 'MODEL', 'FUELTYPE',
       'VEHTYPE', 'WHOMAIN', 'OD_READ', 'HFUEL', 'VEHOWNED', 'VEHOWNMO',
       'ANNMILES', 'HYBRID', 'PERSONID', 'TRAVDAY', 'HOMEOWN', 'HHSIZE',
       'HHVEHCNT', 'HHFAMINC', 'DRVRCNT', 'HHSTATE', 'HHSTFIPS', 'NUMADLT',
       'WRKCOUNT', 'TDAYDATE', 'LIF_CYC', 'MSACAT', 'MSASIZE', 'RAIL', 'URBAN',
       'URBANSIZE', 'URBRUR', 'CENSUS_D', 'CENSUS_R', 'CDIVMSAR', 'HH_RACE',
       'HH_HISP', 'HH_CBSA', 'SMPLSRCE', 'WTHHFIN', 'BESTMILE', 'BEST_FLG',
       'BEST_EDT', 'BEST_OUT', 'HBHUR', 'HTHTNRNT', 'HTPPOPDN', 'HTRESDN',
       'HTEEMPDN', 'HBHTNRNT', 'HBPPOPDN', 'HBRESDN'],
      dtype='object')
```

In [298]:
```
# Column Types
vehpub.dtypes.value_counts()
```

Out[298]:
```
int64      47
object      5
float64     2
dtype: int64
```

Data Cleaning (10 points)

1. Identification of States with High EV Adoption Rates

As previously mentioned, some states are further along the adoption curve than others. As such, we would like to identify which states would make good candidates in the development of the model. To do this, we look to the vehicle data to see which are electric vehicles and in which state the household resides.

It is important to note here that 'WTHHFIN' is the weight of the household representative of the total population. Because vehicles are tied to households rather than individual persons, the household weight will be used.

In [387]:
```
# Quick View
veh_types = vehpub[['HOUSEID','VEHID','HHSTATE','HFUEL','WTHHFIN']]
veh_types.head()
```

Out[387]:

	HOUSEID	VEHID	HHSTATE	HFUEL	WTHHFIN
0	30000007	1	NC	-1	187.31432
1	30000007	2	NC	-1	187.31432
2	30000007	3	NC	-1	187.31432
3	30000007	4	NC	-1	187.31432
4	30000007	5	NC	-1	187.31432

In this project, there are two classifications of vehicles that I considered to be Electric Vehicles - Plug-in Hybrids and full Electrics. This is because these are the types which require infrastructure enhancements. In the step below, I created a binary column that equates to 1 if the vehicle is Electric or Plug-in Hybrid, and 0 otherwise.

In [388]:
```
# HFUEL = 2 (Plug-in Hybrid) and HFUEL = 3 (Electric)
veh_types.loc[:,'EV'] = ((veh_types['HFUEL'] == 2) | (veh_types['HFUEL'] == 3)).astype(int)
```

I then weighted the EV status of each vehicle to the population.

In [389]:
```
veh_types['EV_weighted'] = veh_types['EV']*veh_types['WTHHFIN']
veh_types.head()
```

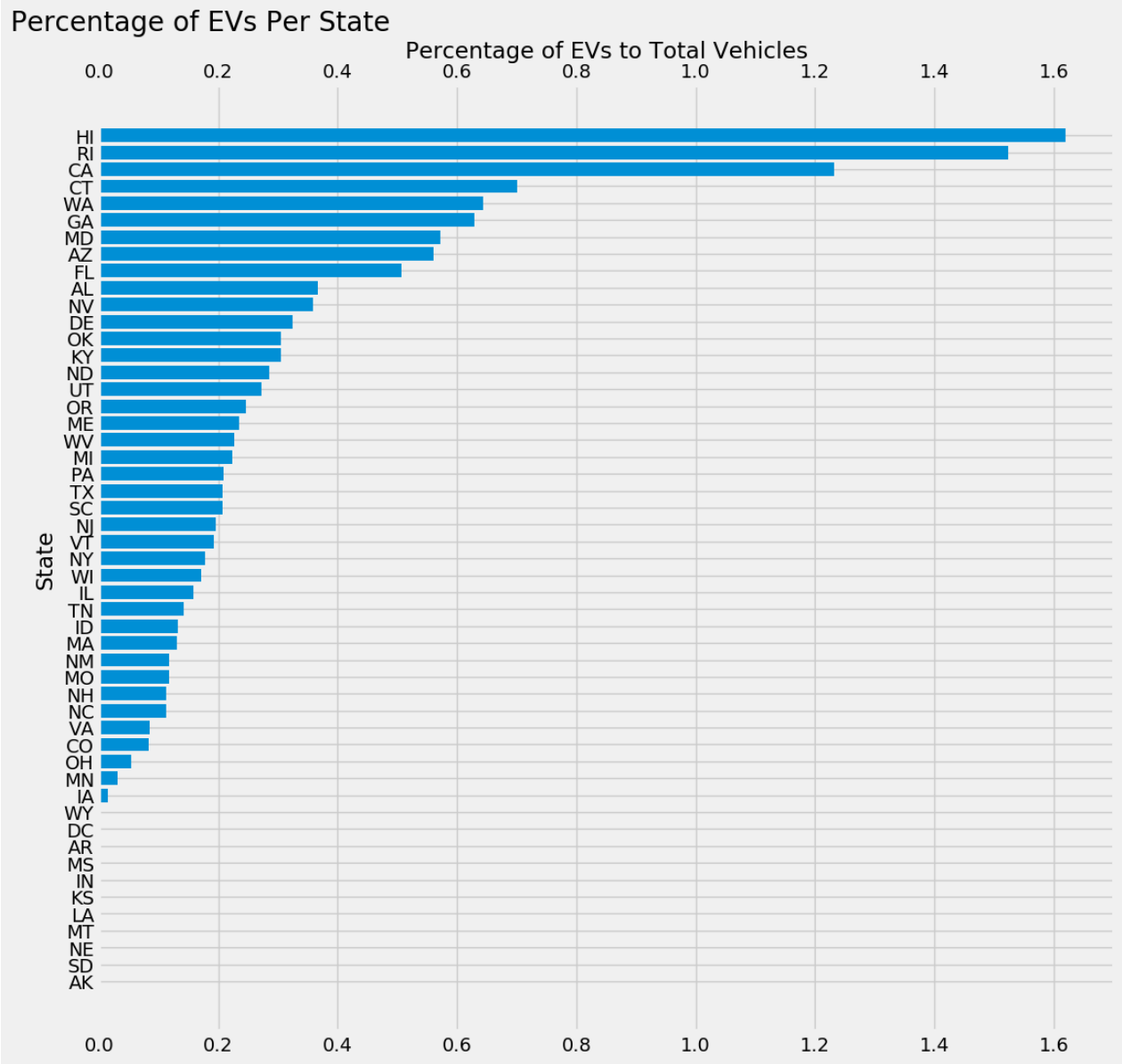
Out[389]:

	HOUSEID	VEHID	HHSTATE	HFUEL	WTHHFIN	EV	EV_weighted
0	30000007	1	NC	-1	187.31432	0	0.0
1	30000007	2	NC	-1	187.31432	0	0.0
2	30000007	3	NC	-1	187.31432	0	0.0
3	30000007	4	NC	-1	187.31432	0	0.0
4	30000007	5	NC	-1	187.31432	0	0.0

In this next step, the ratio of EVs to the total vehicle count per state was calculated and plotted in descending order.

In [390]:
```
# Groupby state, sum EVs
state_evs = veh_types.groupby(by='HHSTATE')[['EV','EV_weighted','WTHHFIN']].sum()
state_evs = state_evs.rename(columns={'EV_weighted':'EV Count','WTHHFIN':'Veh Count'})

# Calculate EV Ratio and sort
state_evs['EV Ratio'] = state_evs['EV Count'] / state_evs['Veh Count'] *100
state_evs = state_evs.sort_values(by=['EV Ratio'],ascending=True)
```

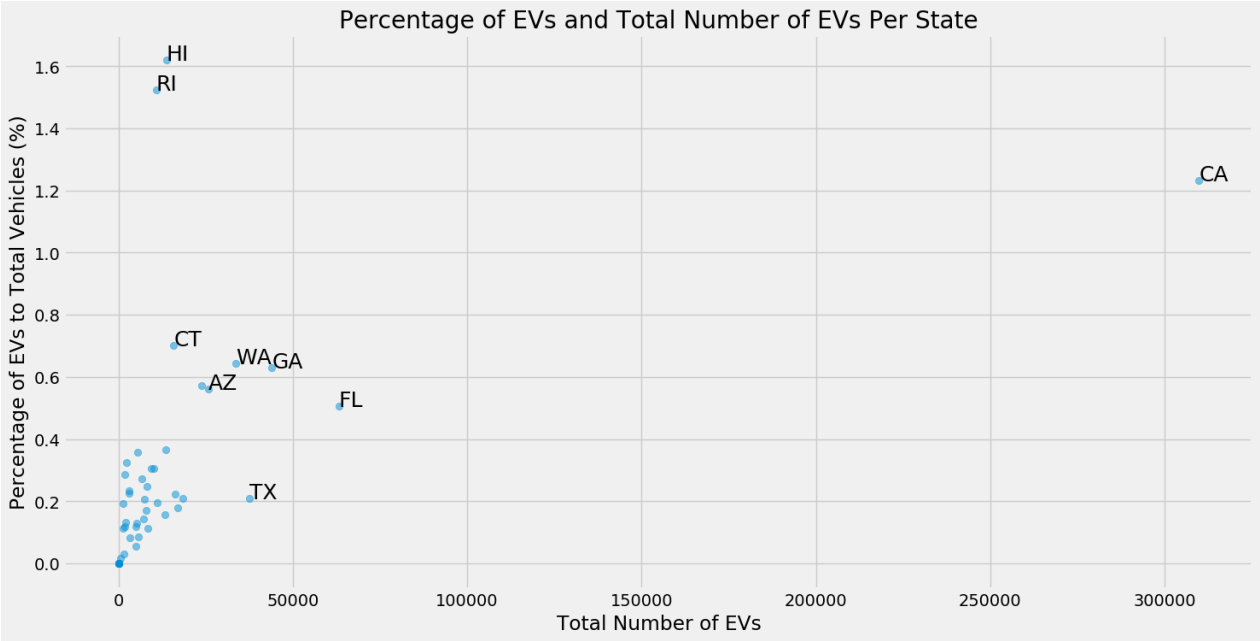
In [391]:
```
fig, ax1 = plt.subplots(figsize=(12,12))
ax1.xaxis.set_tick_params(labeltop='on')

ax1.barh(state_evs.index,state_evs['EV Ratio'])
ax1.set_title('Percentage of EVs Per State', y=1.05, x=0.10, fontsize=20)
ax1.set_ylabel('State')
ax1.set_xlabel('Percentage of EVs to Total Vehicles')
ax1.xaxis.set_label_position('top')
plt.show()
```



Generally, adoption rates are still relatively low. From here, we can see that states like Hawaii, Rhode Island, and California have significantly higher adoption among all the states.

However, let us perform a check to see the raw values of the number of EVs Per State.

In [392]:
```
fig, ax = plt.subplots(figsize=(16,8))
ax.set_title('Percentage of EVs and Total Number of EVs Per State')
ax.set_ylabel('Percentage of EVs to Total Vehicles (%)')
ax.set_xlabel('Total Number of EVs')
ax.set_ybound(lower=0,upper=0.025)
ax.scatter(state_evs['EV Count'],state_evs['EV Ratio'],alpha=0.5)

# Label Prominent States
for i, txt in enumerate(state_evs.index):
    if txt in ['CA','HI','WA','TX','GA','AZ','RI','CT','FL']:
        ax.annotate(txt, (state_evs['EV Count'][i]+3, state_evs['EV Ratio'][i]+0.0001),fontsize
=18)
plt.show()
```



Looking at the numbers:

In [393]:
```
state_evs.loc[['CA','HI','RI'],:]
```

Out[393]:

	EV	EV Count	Veh Count	EV Ratio
HHSTATE				
CA	584	309837.900705	2.513810e+07	1.232543
HI	10	13681.909785	8.446364e+05	1.619858
RI	2	10815.898456	7.100959e+05	1.523160

Although Hawaii and Rhode Island have relatively high adoption rates, the amount of data available is very low - 10 data points accounting for 13,682 EVs and 2 data points accounting for 10,816 EVs, respectively. On the other hand, California has 584 data points accounting for 309,838 EVs. As such, the model will be created using the California data.

Later on, the model will be used to predict potential EV ownership in another state. For this project, I've arbitrarily selected Texas, and will be part of the data cleaning process.

In [394]:
```
# Household Information
hhpub_ca = hhpub[hhpub['HHSTATE'] == 'CA'].reset_index(drop=True)
hhpub_tx = hhpub[hhpub['HHSTATE'] == 'TX'].reset_index(drop=True)

# Vehicle Information
vehpub_ca = vehpub[vehpub['HHSTATE'] == 'CA'].reset_index(drop=True)
vehpub_tx = vehpub[vehpub['HHSTATE'] == 'TX'].reset_index(drop=True)
```

As a check, for the households that own EVs, I obtained the distribution of the number of EVs they own.

Like what I did earlier, I created a new dataframe for this task with the features I need and added a binary column 'EV', which gives 1 if the vehicle is an EV or Plug-in, and 0 otherwise.

In [463]:
```
# Just the columns needed
veh_types_ca = vehpub_ca[['HOUSEID','HFUEL','WTHHFIN']]
veh_types_tx = vehpub_tx[['HOUSEID','HFUEL','WTHHFIN']]

# Binary column: 1 if EV or Plug-In Hybrid, 0 otherwise
veh_types_ca.loc[:,'EV'] = ((veh_types_ca['HFUEL'] == 2) | (veh_types_ca['HFUEL'] == 3)).astype
(int)
veh_types_tx.loc[:,'EV'] = ((veh_types_tx['HFUEL'] == 2) | (veh_types_tx['HFUEL'] == 3)).astype
(int)
```

Next, I grouped the vehicles by Household ID and summed the EVs for that household. The household weight 'WTHHFIN' is the same for all vehicles of the same household, so I implemented the 'mean' function to retain its value. I then grouped it by the number of EVs per household. Effectively, this is congruent to value_counts weighted by 'WTHHFIN'.

In [464]:
```
# Determine Number of EVs Per Household
hh_ev_ca = veh_types_ca.groupby(by='HOUSEID').agg({'EV':'sum','WTHHFIN':'mean'})
hh_ev_tx = veh_types_tx.groupby(by='HOUSEID').agg({'EV':'sum','WTHHFIN':'mean'})

# Determine value counts of EVs per Household, with Household weight applied
hh_ev_ca_dist = hh_ev_ca.groupby(by='EV').sum()[1:]
hh_ev_tx_dist = hh_ev_tx.groupby(by='EV').sum()[1:]

# Solve for the Distribution
hh_ev_ca_dist['Distribution%'] = hh_ev_ca_dist['WTHHFIN'] / np.sum(hh_ev_ca_dist['WTHHFIN']) *
100
hh_ev_tx_dist['Distribution%'] = hh_ev_tx_dist['WTHHFIN'] / np.sum(hh_ev_tx_dist['WTHHFIN']) *
100

hh_ev_ca_dist
```

Out[464]:

	WTHHFIN	Distribution%
EV		
1	284810.297076	95.913012
2	11380.899483	3.832644
3	755.268221	0.254345

This means that for the households that own EVs, 95.91% own 1 EV, 3.83% own 2 EVs, and 0.25% own 3 EVs. Because of this distribution, this project will only aim to predict EV ownership per household, regardless of the quantity they own.

Next, it would be useful to find out what percentage of households own an EV.

In [465]:
```
sum(hh_ev_ca_dist['WTHHFIN']) / sum(hh_ev_ca['WTHHFIN']) *100
```

Out[465]: 2.4897303286441548

Based on this, only 2.49% of the households in California own an electric vehicle. Because this number is so small, it is important to select the metric most applicable to perform statistical measures to predict this. For instance, if a model predicts that all households don't own an electric vehicle, its error rate would only be at 2.49%. This will be discussed further when we build the model.

2. Cleaning the variables

To prepare for classification work, a new column is created called EV_bin, where '0' means the household does not own an EV, and '1' means the household owns any number of EVs. As a note, this so far only accounts for households that own vehicles. Households that don't own vehicles aren't part of this set. This will be addressed later on.

In [466]:
```
# 1 if the household owns at least 1 EV, 0 otherwise
hh_ev_ca['EV_bin'] = (hh_ev_ca['EV'] > 0).astype(int)
hh_ev_tx['EV_bin'] = (hh_ev_tx['EV'] > 0).astype(int)
hh_ev_ca.head()
```

Out[466]:

	EV	WTHHFIN	EV_bin
HOUSEID			
30000041	0	788.614240	0
30000085	0	190.669041	0
30000094	0	163.382292	0
30000155	0	120.772451	0
30000227	0	62.015790	0

To consolidate the different datasets, the frame of reference to be used is households. The weight for each household will also be important to keep in mind.

HOUSEID: Household Identifier
WTHHFIN: Final HH weight

Below, variables are either kept or removed based on their appropriateness as predictors. Some columns are removed for this exercise simply to reduce the complexity of the data, which all have to be cleaned or processed. Data could be removed because it is not relevant to the predictive work or because there are multiple features with overlapping scope. Admittedly, this process can be subjective.

(C) - Encoded as categorical
(N) - Encoded as numerical

VARIABLES KEPT

(C) BIKE: Frequency of Bicycle Use for Travel
(C) BIKE2SAVE: Bicycle to Reduce Financial Burden of Travel
(C) BUS: Frequency of Bus Use for Travel
(C) CAR: Frequency of Personal Vehicle Use for Travel
(N) CNTTDHH: Count of household trips on travel day
(N) DRVRCNT: Number of drivers in household
(C) HBHTNRNT: Category of the percent of renter-occupied housing in the census block group of the household's home location
(C) HBPPOPDN: Category of population density (persons per square mile) in the census block group of the household's home location
(C) HBRESDN: Category of housing units per square mile in the census block group of the household's home location
(C) HHFAMINC: Household income
(N) HHSIZE: Count of household members
(N) HHVEHCNT: Count of household vehicles
(C) HOMEOWN: Home Ownership
(C) HTHTNRNT: Category of the percent of renter-occupied housing in the census tract of the household's home location
(C) HTPPOPDN: Category of population density (persons per square mile) in the census tract of the household's home location
(C) HTRESDN: Category of housing units per square mile in the census tract of the household's home location
(N) NUMADLT: Count of adult household members at least 18 years old
(C) PC: Frequency of Desktop or Laptop Computer Use to Access the Internet
(C) PLACE: Travel is a Financial Burden
(C) PRICE: Price of Gasoline Affects Travel
(C) PTRANS: Public Transportation to Reduce Financial Burden of Travel
(C) SPHONE: Frequency of Smartphone Use to Access the Internet
(C) TAB: Frequency of Tablet Use to Access the Internet
(C) TAXI: Frequency of Taxi Service or Rideshare Use for Travel
(C) TRAIN: Frequency of Train Use for Travel
(C) URBANSIZE: Urban area size where home address is located
(C) WALK: Frequency of Walking for Travel
(C) WALK2SAVE: Walk to Reduce Financial Burden of Travel (C) WEBUSE17: Frequency of internet use
(N) WRKCOUNT: Number of workers in household
(N) YOUNGCHILD: Count of persons with an age between 0 and 4 in household

VARIABLES DROPPED

(C) CDIVMSAR: Grouping of household by combination of census division, MSA status, and presence of a subway system when population greater than 1 million
(C) CENSUS_D: 2010 Census division classification for the respondent's home address
(C) CENSUS_R: Census region classification for home address
(C) HH_RACE: Race of household respondent
(C) HBHUR: Urban / Rural indicator - Block group
(C) HHRELATD: At least two household persons are related
(C) HHRESP: Person identifier of household respondent
(C) HHSTATE: Household state
(C) HHSTFIPS: State FIPS for Household address
(C) HH_CBSA: Core Based Statistical Area (CBSA) FIPS code for the respondent's home address
(C) HH_HISP: Hispanic status of household respondent
(C) HTEEMPDN: Category of workers per square mile in the census tract of the household's home location
(C) LIF_CYC: Life Cycle classification for the household, derived by attributes pertaining to age, relationship, and work status.
(C) MSACAT: Metropolitan Statistical Area (MSA) category for the household's home address, based on household's home geocode and TIGER/Line Shapefiles.
(C) MSASIZE: Population size category of the Metropolitan Statistical Area (MSA), from the 2010-2014 five-year American Community Survey (ACS) API.
(C) PARA: Frequency of Paratransit Use for Travel
(C) RAIL: MSA heavy rail status for household
(N) RESP_CNT: Count of responding persons per household
(C) SAMPSTRAT: Primary Sampling Stratum Assignment
(C) SCRESP: Person identifier of mail screener respondent, always 1 to roster self first
(C) SMPLSRCE: Sample where the case originated
(C) TDAYDATE: Date of travel day (YYYYMM)

(C) TRAVDAY: Travel day - day of week
(C) URBAN: Household's urban area classification, based on home address and 2014 TIGER/Line Shapefile

In [467]:
```
model_df = hhpub_ca.set_index('HOUSEID')
predict_df = hhpub_tx.set_index('HOUSEID')
```

The target variable prepared earlier is added to this dataframe. As mentioned, that did not account for households that did not own any vehicle, and NaN values are placed in their rows. As such, these NaN values are replaced with zero.

In [491]:
```
# Import the classification determined earlier
model_df['EV_bin'] = hh_ev_ca['EV_bin']
predict_df['EV_bin'] = hh_ev_tx['EV_bin']

# Nan values appear for households that don't own vehicles at all. Replace these with zero.
model_df['EV_bin'] = model_df['EV_bin'].fillna(0)
predict_df['EV_bin'] = predict_df['EV_bin'].fillna(0)

# Quick View
model_df.head()
```

Out[491]:

	TRAVDAY	SAMPSTRAT	HOMEOWN	HHSIZE	HHVEHCNT	HHFAMINC	PC	SPHONE	TAB	WALK	...	W
HOUSEID												
30000041	4	3	1	2	2	11	1	1	1	1	...	788
30000085	1	2	1	1	2	9	1	1	4	1	...	190
30000094	3	3	2	1	1	4	1	1	5	4	...	163
30000155	1	1	1	1	2	-7	1	5	1	5	...	120
30000227	1	2	1	2	2	6	1	5	1	-9	...	62.

5 rows × 58 columns

To ease data cleaning, the features remaining were organized based on their nature and structure.

Numerical Data: 'CNTTDHH','DRVRCNT','HHSIZE','HHVEHCNT','NUMADULT','WRKCOUNT','YOUNGCHILD'
Frequency Data: 'BIKE','BUS','CAR','PC','SPHONE','TAB','TAXI','TRAIN','WALK','WEBUSE17'
Range-Mean Data: 'HBHTNRNT','HBPPOPDN','HBRESDN','HTHTNRNT','HTPPOPDN','HTRESDN'
Range-Map Data: 'HHFAMINC','URBANSIZE'
Categorical Data: 'HOMEOWN'
Opinion Data: 'PLACE','PRICE','PTRANS'

I then transformed the data per type. As the features are transformed, they will be merged consecutively to form the final features DataFrame.

In [492]:
```
features = pd.DataFrame()
features_tx = pd.DataFrame()
```

Numerical Data is ready to be used and can be added to the final DataFrame.

In [493]:
```
features = features.append(model_df[['CNTTDHH','DRVRCNT','HHSIZE','HHVEHCNT','NUMADLT','WRKCOUN
T','YOUNGCHILD']])
features_tx = features_tx.append(predict_df[['CNTTDHH','DRVRCNT','HHSIZE','HHVEHCNT','NUMADLT',
'WRKCOUNT','YOUNGCHILD']])
```

Frequency Data is loosely tied to ordered values (daily, a few times a week, a few times a month, a few times a year, never, others). However, because the values still provide a quantitative index of frequency, they are treated as quantitative variables. Three types of answers, '-9=Not ascertained, -8=I don't know, and -7=I prefer not to answer are removed because they cannot be used for the model.

In [494]:
```
freq_var = ['BIKE','BUS','CAR','PC','SPHONE','TAB','TAXI','TRAIN','WALK','WEBUSE17']

for var in freq_var:
    # CA
    var_df = pd.DataFrame(model_df[var][model_df[var] > 0])
    features = features.merge(var_df,left_index=True,right_index=True)
    # TX
    var_df_tx = pd.DataFrame(predict_df[var][predict_df[var] > 0])
    features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)
```

It is important to take note of how much of the data was lost in this process.

In [495]:
```
print(len(model_df), '->', len(features))
```

```
26099 -> 22024
```

Range-Mean Data refers to ranges of values (mostly) binned to the mean of the range. For instance, if the range is a percentage value of 45-54%, the value assigned is 50. Because the ordering of value here is very clear, these will be used as numerical variables. Some values need to be adjusted to better fit this format. Currently, 0=0-4%, 5=5-14%, and 95=95-100%.

There is a very small number of answers that have the value -9=Not ascertained. Because these are few in number and have no numerical value, these records will be removed from the model.

In [496]:
```
range_mean_var = ['HBHTNRNT','HBPPOPDN','HBRESDN','HTHTNRNT','HTPPOPDN','HTRESDN']

for var in range_mean_var:
    # CA
    model_df[var][model_df[var] == 5] = 10
    model_df[var][model_df[var] == 0] = 2
    model_df[var][model_df[var] == 95] = 97
    var_df = pd.DataFrame(model_df[var][model_df[var] > 0])
    features = features.merge(var_df,left_index=True,right_index=True)
    # TX
    predict_df[var][predict_df[var] == 5] = 10
    predict_df[var][predict_df[var] == 0] = 2
    predict_df[var][predict_df[var] == 95] = 97
    var_df_tx = pd.DataFrame(predict_df[var][predict_df[var] > 0])
    features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)
```

Note that 3 datapoints were removed due to this.

In [497]:
```
print(len(model_df), '->', len(features))
```

```
26099 -> 22021
```

Range-Map Data refers to ranges of values mapped to an order of numbers. For instance, for a range of household income from 15,000 to 24,999, the value assigned is 3. However, the ranges of the intervals are not equal to each other. Hence, the values assigned are not linearly proportional to the income range. Despite this, it is reasonable to treat these variables as numerical variables as they map quantitative values where the order may be relevant to the prediction. Like in the previous section, values with -9=Not ascertained, -8=I don't know, and -7=I prefer not to answer will be removed from the dataset.

In [498]:
```
# Household Income
# CA
var_df = pd.DataFrame(model_df['HHFAMINC'][model_df['HHFAMINC'] > 0])
features = features.merge(var_df,left_index=True,right_index=True)
# TX
var_df_tx = pd.DataFrame(predict_df['HHFAMINC'][predict_df['HHFAMINC'] > 0])
features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)

# Urbanization Category
# CA
var_df = pd.DataFrame(model_df['URBANSIZE'])
var_df[var_df == 6] = 0
features = features.merge(var_df,left_index=True,right_index=True)
# CA
var_df_tx = pd.DataFrame(predict_df['URBANSIZE'])
var_df_tx[var_df_tx == 6] = 0
features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)
```

The size of the data has been further reduced.

In [499]:
```
print(len(model_df), '->', len(features))
```

```
26099 -> 21393
```

Categorical Data are those that are explicitly categorical, such as whether the household is in an urban, suburban, or rural census block. Similar to what was done with the frequency data, one-hot encoding will be implemented for these variables. Once again, miscellaneous answers will be grouped into an 'Other' category, which will be excluded from the features table due to mutual exclusivity.

In [500]:
```
# CA
model_df['HOMEOWN'][(model_df['HOMEOWN'] < 0) | (model_df['HOMEOWN'] == 97)] = 0
var_df = pd.get_dummies(model_df['HOMEOWN'],drop_first=True,prefix='HOMEOWN')
features = features.merge(var_df,left_index=True,right_index=True)

# TX
predict_df['HOMEOWN'][(predict_df['HOMEOWN'] < 0) | (predict_df['HOMEOWN'] == 97)] = 0
var_df_tx = pd.get_dummies(predict_df['HOMEOWN'],drop_first=True,prefix='HOMEOWN')
features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)
```

Opinion Data are those that ask the respondent about his opinions on certain statements, with answers ranging from strongly disagree to strongly agree. Because the assigned values have an order to them that reflect agreeableness to the statement, these data can be treated as numerical categories. Like before, miscellaneous answers, -9=Not ascertained, -8=I don't know, and -7=I prefer not to answer will be removed from the data set.

In [501]:
```
opinion_var = ['BIKE2SAVE','PLACE','PRICE','PTRANS','WALK2SAVE']

for var in opinion_var:
    # CA
    var_df = pd.DataFrame(model_df[var][model_df[var] > 0])
    features = features.merge(var_df,left_index=True,right_index=True)
    # TX
    var_df_tx = pd.DataFrame(predict_df[var][predict_df[var] > 0])
    features_tx = features_tx.merge(var_df_tx,left_index=True,right_index=True)
```

In [502]:
```
print(len(model_df), '->', len(features))
```

```
26099 -> 21037
```

Let us now take a look at the features of the model.

In [556]:
```
features.columns
```

Out[556]:
```
Index(['CNTTDHH', 'DRVRCNT', 'HHSIZE', 'HHVEHCNT', 'NUMADLT', 'WRKCOUNT',
       'YOUNGCHILD', 'BIKE', 'BUS', 'CAR', 'PC', 'SPHONE', 'TAB', 'TAXI',
       'TRAIN', 'WALK', 'WEBUSE17', 'HBHTNRNT', 'HBPPOPDN', 'HBRESDN',
       'HTHTNRNT', 'HTPPOPDN', 'HTRESDN', 'HHFAMINC', 'URBANSIZE', 'HOMEOWN_1',
       'HOMEOWN_2', 'BIKE2SAVE', 'PLACE', 'PRICE', 'PTRANS', 'WALK2SAVE'],
      dtype='object')
```

Let us now set up the target variable, accounting for the data points that were removed earlier.

In [504]:
```
target = model_df.loc[list(features.index),'EV_bin']
target_tx = predict_df.loc[list(features_tx.index),['EV_bin','WTHHFIN']]
```

Data Summary and Exploratory Data Analysis (10 points)

First, I tried to get a general sense of the distribution of the features. With the tradeoff of some level of abstraction, I decided to start with split violinplots for each variable. However, I limited the features to visualize as it would be very difficult to compare 32 plots side by side. To select features to visualize, one was chosen for each type of data listed earlier (Numerical, Range-Mean, Categorical, etc.)

On one side of the violinplot in blue are the distributions of households who don't own electric vehicles; in orange are the distributions of households who own electric vehicles.

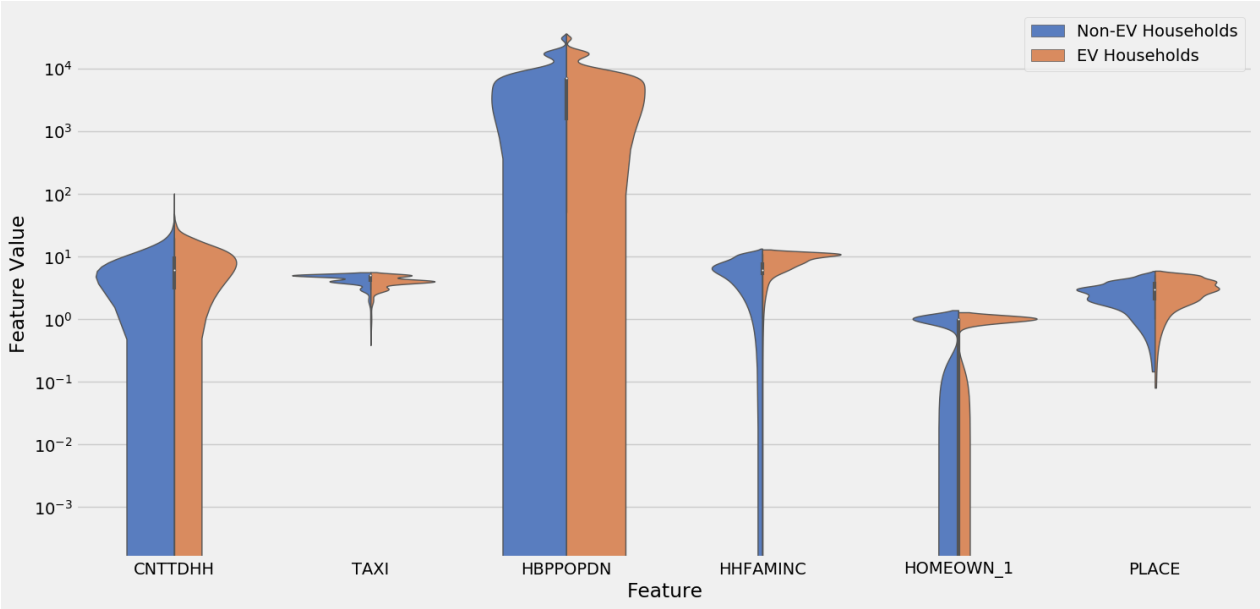
The code below is a slightly complicated preparatory step for the data to be used on seaborn's violin plots. You can also view the structure of the DataFrame needed as input to the violinplot function.

In [554]:
```
vis_dist_df = features.stack().reset_index(level=1, name='Feature Value')\
                              .rename(columns={'level_1':'Feature'})[['Feature','Feature Value'
]]
vis_dist_df = pd.merge(vis_dist_df,pd.DataFrame(target),left_index=True,right_index=True)
vis_dist_df.head()
```

Out[554]:

	Feature	Feature Value	EV_bin
HOUSEID			
30000041	CNTTDHH	6	0.0
30000041	DRVRCNT	2	0.0
30000041	HHSIZE	2	0.0
30000041	HHVEHCNT	2	0.0
30000041	NUMADLT	2	0.0

In [585]:
```
plt.figure(figsize=(16,8))
ax = sns.violinplot(x="Feature",
                    y="Feature Value",
                    hue="EV_bin",
                    data=vis_dist_df,
                    order=['CNTTDHH','TAXI','HBPPOPDN','HHFAMINC','HOMEOWN_1','PLACE'],
                    linewidth=1,
                    palette="muted",
                    split=True,
                    bw=0.4)
ax.figure.get_axes()[0].set_yscale('log')
# Set Legend
handles, labels = ax.figure.get_axes()[0].get_legend_handles_labels()
ax.figure.get_axes()[0].legend(handles, ['Non-EV Households','EV Households'])
plt.show()
```

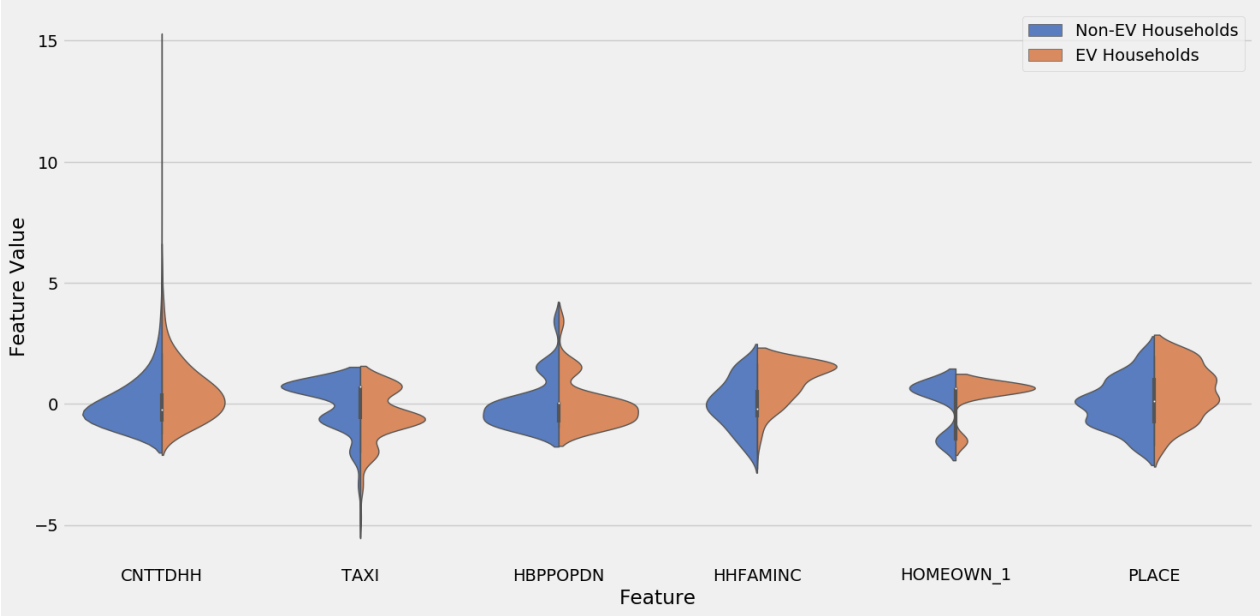


Because of the large differences in range, I decided to plot the y axis on logarithmic scale. Although this also warps the distribution, we can make some insights from the plot. Most importantly, the difference in ranges across different features is significant, with features like 'HBPPOPDN' (population density of the census block) reaching values above 10,000 and features like 'TAXI' (frequency of taxi use) not making it to values of 10. HOMEOWN_1 here looks weird because it's a categorical variable with values of only 0 or 1. This means that standardization of the variables may be needed later for some models - in particular, regression models like Ordinary Least Squares, Lasso, and Ridge.

I went ahead and built a standardized features matrix now, which could be useful later on. I then repeated the steps above to produce another set of violin plots.

In [578]:
```
from sklearn.preprocessing import scale
features_norm = pd.DataFrame(scale(features),columns=features.columns,index=features.index)
```

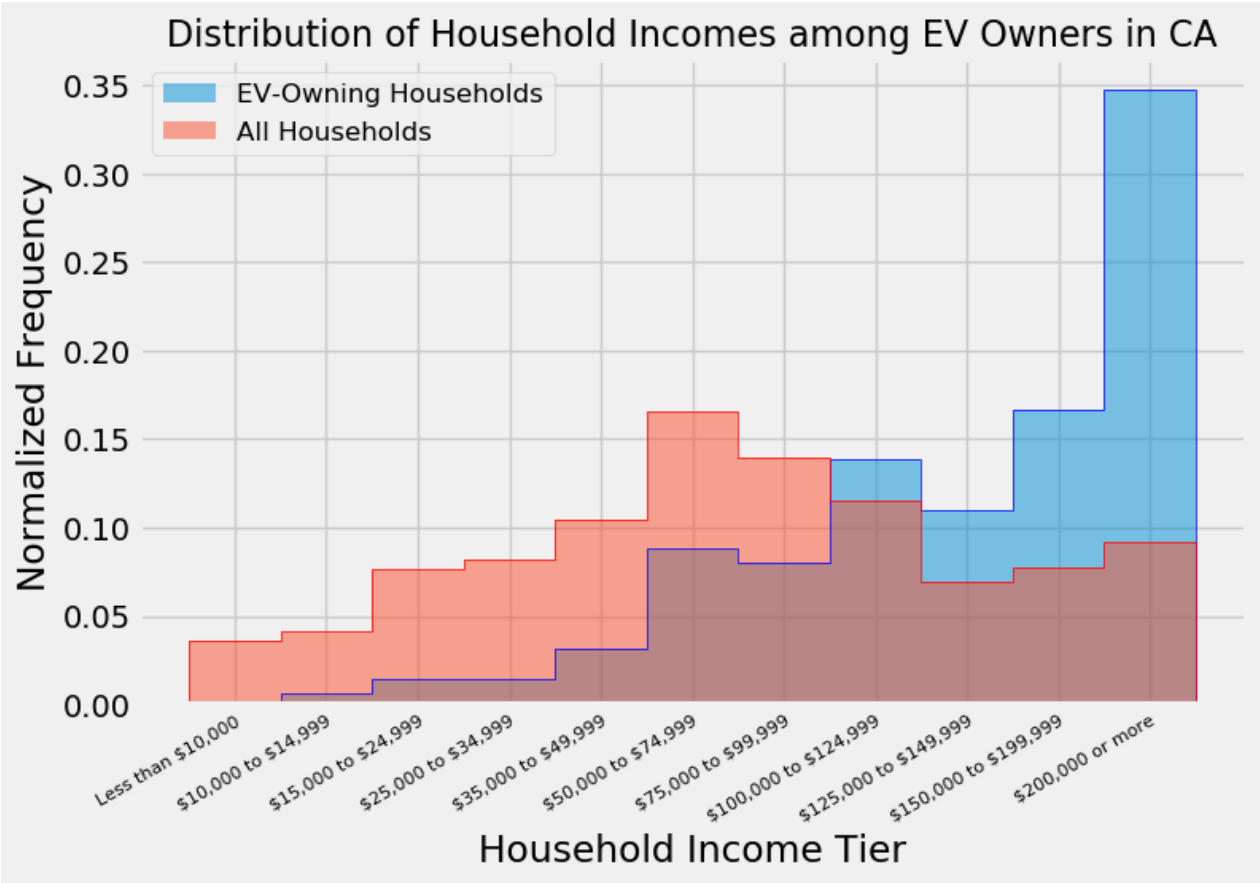
In [586]:
```
vis_dist_norm_df = features_norm.stack().reset_index(level=1, name='Feature Value')\
                                .rename(columns={'level_1':'Feature'})[['Feature','Feature Value'
]]
vis_dist_norm_df = pd.merge(vis_dist_norm_df,pd.DataFrame(target),left_index=True,right_index=T
rue)
vis_dist_norm_df
plt.figure(figsize=(16,8))
ax = sns.violinplot(x="Feature",
                    y="Feature Value",
                    hue="EV_bin",
                    data=vis_dist_norm_df,
                    order=['CNTTDHH','TAXI','HBPPOPDN','HHFAMINC','HOMEOWN_1','PLACE'],
                    linewidth=1,
                    palette="muted",
                    split=True,
                    bw=0.4)
# Set Legend
handles, labels = ax.figure.get_axes()[0].get_legend_handles_labels()
ax.figure.get_axes()[0].legend(handles, ['Non-EV Households','EV Households'])
plt.show()
```



This looks much better - note that the y axis is no longer on a logarithmic scale. However, it does not necessarily mean that the normalized set of data will perform better for the models. It is important to note that some variables like 'CNTTDHH' have extreme outliers that could have affected the standardization process. The only way to know if normalization improves the model is to test it out later.

In the project background, I mentioned that wealthier households may be more likely to purchase EVs because of their high cost. This is slightly visible in the violin plot above under the variable 'HHFAMINC' (household income). However, let us perform a deeper dive with more details to see the distribution of household income among EV owners in California.

In [414]:
```
fig, ax = plt.subplots(figsize=(8,5))

# Normalized Histogram Plots
ax.hist(features[target == 1]['HHFAMINC'],range=(0.5,11.5),bins=11,normed=True,alpha=0.5)
ax.hist(features['HHFAMINC'],range=(0.5,11.5),bins=11,normed=True,alpha=0.5)

# Just for a nice outline
ax.hist(features[target == 1]['HHFAMINC'],histtype='step',color='b',range=(0.5,11.5),bins=11,no
rmed=True)
ax.hist(features['HHFAMINC'],histtype='step',color='r',range=(0.5,11.5),bins=11,normed=True)

plt.title('Distribution of Household Incomes among EV Owners in CA',fontsize=16)
plt.xlabel('Household Income Tier')
plt.ylabel('Normalized Frequency')
ax.set_xticks(np.arange(1,12))
ax.set_xticklabels(labels=['Less than \$10,000',
                           '\$10,000 to \$14,999',
                           '\$15,000 to \$24,999',
                           '\$25,000 to \$34,999',
                           '\$35,000 to \$49,999',
                           '\$50,000 to \$74,999',
                           '\$75,000 to \$99,999',
                           '\$100,000 to \$124,999',
                           '\$125,000 to \$149,999',
                           '\$150,000 to \$199,999',
                           '\$200,000 or more'],
                   rotation=30,
                   ha='right',
                   rotation_mode='anchor',
                   fontsize=8)
plt.legend(['EV-Owning Households','All Households'],fontsize=12)
plt.show()
```

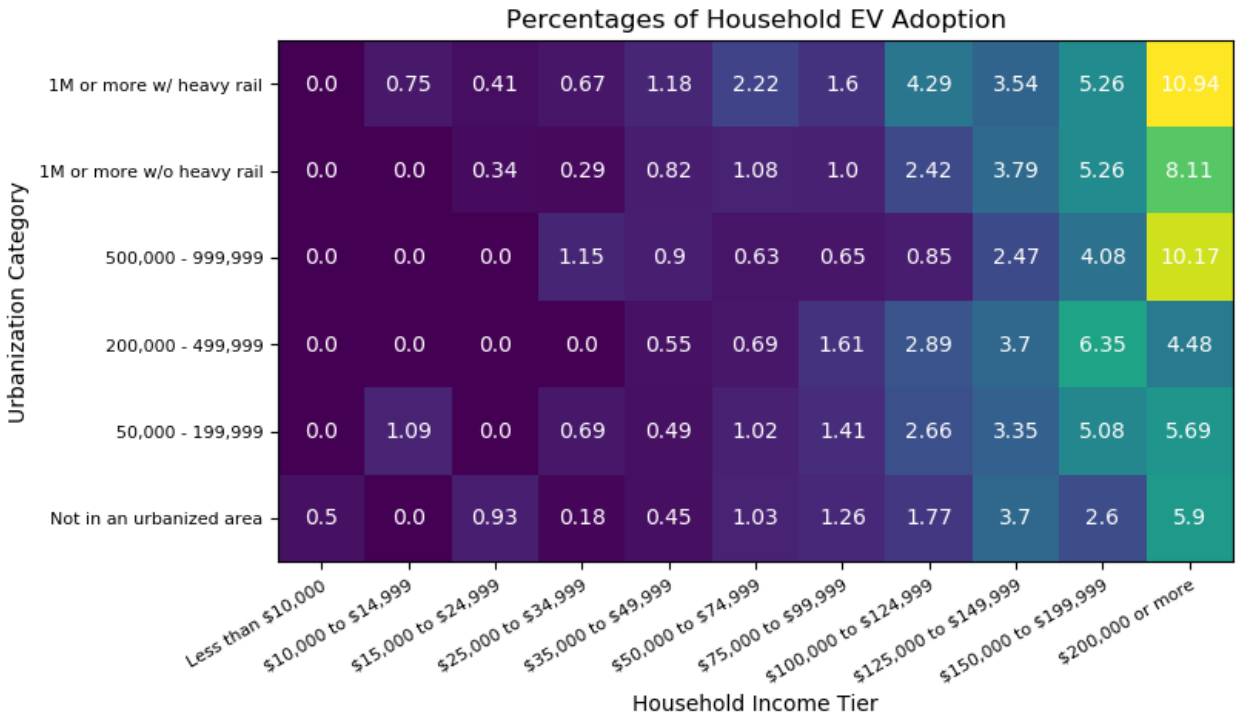


While the general population of California is normally distributed, the households that own electric vehicles have a very heavy skew towards higher income tiers.

I then tried to pair this with another feature that could be relevant to predicting EV ownership in a household. Because electric vehicles are forms of transportation, perhaps some geographical features could be a factor to ownership. Because most of the data available are organized into bins, it is difficult to plot what would have been continuous data in real life. Instead, I plotted a categorical heatmap of EV ownership of households against their income tier and the urbanization category where the home is located.

In [415]:
```
# Feature Setting
x_view = features['HHFAMINC']  # Set x as household income
y_view = features['URBANSIZE'] # Set y as urbanization category

# Transforming the Data
ev_table = pd.crosstab(y_view[target == 1],x_view[target == 1]) # All households that own EVs
hh_table = pd.crosstab(y_view,x_view)                           # All households
ratio_table = ev_table / hh_table * 100                         # Ratio of EV-owning to non-EV-
owning
ratio_table = np.round(ratio_table.fillna(0),decimals=2)        # Replace NaN with 0 in case of
 0 denominators
```

In [416]:
```
plt.style.use('default')

fig, ax = plt.subplots(figsize=(8,8))
im = ax.imshow(ratio_table,origin='lower')

# Show values in each cell
for i in range(y_view.nunique()):
    for j in range(x_view.nunique()):
        text = ax.text(j, i, ratio_table.iloc[i, j],
                       ha="center", va="center", color="w")

ax.set_xticks(np.arange(x_view.nunique()))
ax.set_yticks(np.arange(y_view.nunique()))
ax.set_xticklabels(labels=['Less than \$10,000',
                           '\$10,000 to \$14,999',
                           '\$15,000 to \$24,999',
                           '\$25,000 to \$34,999',
                           '\$35,000 to \$49,999',
                           '\$50,000 to \$74,999',
                           '\$75,000 to \$99,999',
                           '\$100,000 to \$124,999',
                           '\$125,000 to \$149,999',
                           '\$150,000 to \$199,999',
                           '\$200,000 or more'],
                   rotation=30,
                   ha='right',
                   rotation_mode='anchor',
                   fontsize=8)
ax.set_yticklabels(labels=['Not in an urbanized area',
                           '50,000 - 199,999',
                           '200,000 - 499,999',
                           '500,000 - 999,999',
                           '1M or more w/o heavy rail',
                           '1M or more w/ heavy rail'],
                   fontsize=8)

ax.set_title("Percentages of Household EV Adoption")
ax.set_xlabel('Household Income Tier')
ax.set_ylabel('Urbanization Category')

plt.show()

#Revert to Old Style Afterwards
plt.style.use('fivethirtyeight')
```



In the plot above, we can see that aside from the trend previously observed for household incomes, there is also a slight trend related to urbanization. Households located in more urbanized settings seem to be more likely to own an electric vehicle.

Forecasting and Prediction Modeling (25 points)

To start, I first created a test split of 20%. This set of data will not be used in the formulation of any model, but will be used to compare the performances of different models to each other. Just to ensure repeatability of results, I set the random_state to a value of 1.

In [587]:
```
from sklearn.model_selection import train_test_split

# split test set
X, X_test, y, y_test = train_test_split(features,target,test_size=0.20,random_state=1)
X_norm, X_test_norm, y, y_test = train_test_split(features_norm,target,test_size=0.20,random_st
ate=1)
```

After that, I then split the data again to create the training set, which the models will be based on, and the validation set, which will help tune each model.

In [588]:
```
# split between train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X,y,test_size=0.25,random_state=1)
X_train_norm, X_val_norm, y_train, y_val = train_test_split(X_norm,y,test_size=0.25,random_stat
e=1)
```

Before proceeding further, I mentioned in an earlier section that the metric to be used to evaluate models has to be appropriately selected because of the imbalance of classes in the data. Upon further research, the metric to be implemented is Precision-Recall (https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html), which deals well with heavy imbalance.

Instead of evaluating the mean squared error of the results as a whole, precision and recall measure result relevancy. To explain this, the precision and recall scores are computed as follows:

$P = \frac{T_p}{T_p+F_p}$

$R = \frac{T_p}{T_p+F_n}$

Similar to the bias-variance tradeoff, there is also a tradeoff between Precision and Recall. According to the scikit-learn documentation linked above:

> A system with high recall but low precision returns many results, but most of its predicted labels are incorrect when compared to the training labels. A system with high precision but low recall is just the opposite, returning very few results, but most of its predicted labels are correct when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly.

In order to balance these, the score I will be using for evaluation in this project is the F1 Score or F Score, which is the harmonic mean of precision and recall:

$F = 2\frac{P \times R}{P+R}$

In [419]:
```
from sklearn.metrics import f1_score
```

Furthermore, another aspect I introduced to my modeling process is the adjustment of the threshold in which a household will be predicted as Class 0 (does not own EVs) or Class 1 (owns EVs). Ordinarily, this is set to 0.5, with good reason. If the probability tilts to Class 1, it's more likely to be Class 1, and vice versa. However, due to the imbalance of classes in my dataset, Class 0 tends to overwhelm Class 1, such that models will always predict Class 0, even for the households with the highest probability to be Class 1.

Now that is set, I created some functions that will be used repeatedly throughout the modeling process. Brief descriptions are provided for each function.

In [420]:
```
def determine_fscores(y_pred):
    """
    Computes the Macro F Score, and F Scores for Class 0 and Class 1 separately.
    """
    # Non-weighted Mean F Score
    fscore_macro = f1_score(y_val,y_pred,average='macro')
    # Separate F Scores per Class
    fscore = f1_score(y_val,y_pred,average=None)
    return fscore_macro, fscore[0], fscore[1]

def determine_threshold(y_prob,thresholds):
    """
    Computes the Macro F Score, and F Scores for Class 0 and Class 1 for differing thresholds.
    """
    # Initialize
    fscores_macro = []
    fscores_0 = []
    fscores_1 = []

    for threshold in thresholds:

        # Predict Class (0 or 1) based on different thresholds
        y_pred = (y_prob >= threshold).astype(int)

        fscore_macro, fscore_0, fscore_1 = determine_fscores(y_pred)

        # Non-weighted Mean F Score
        fscores_macro.append(f1_score(y_val,y_pred,average='macro'))

        # Separate F Scores per Class
        fscores_0.append(fscore_0)
        fscores_1.append(fscore_1)

    return thresholds, fscores_macro, fscores_0, fscores_1

def plot_threshold_fscores(threshhold_vals,model_name):
    """
    Plots the Macro F Score, and F Scores for Class 0 and Class 1 for differing thresholds.
    """
    thresholds, fscores_macro, fscores_0, fscores_1 = threshold_vals
    plt.figure(figsize=(8,5))

    # Plots
    plt.plot(thresholds,fscores_0,label='Predict Households w/o EVs')
    plt.plot(thresholds,fscores_1,label='Predict Households w/ EVs')
    plt.plot(thresholds,fscores_macro,label='Non-weighted Mean F Score',linestyle='dotted')
    plt.vlines(thresholds[fscores_macro.index(max(fscores_macro))],0,1,color='k',linewidth=1,zo
rder=3)

    # Labels
    plt.title('F Scores for ' + model_name)
    plt.xlabel('Thresholds (Default=0.5)')
    plt.ylabel('F Score')
    plt.legend(loc=(0.55,0.59),fontsize=12)

    plt.show()

def determine_optimum_threshold(threshhold_vals,print_vals=True):
    """
    Determines the Optimum threshold based on the highest Macro F Score.
    """
    thresholds, fscores_macro, fscores_0, fscores_1 = threshold_vals
    # Determine F Scores at Optimum Threshold
    opt_fscore_max = max(fscores_macro)
    best_threshold_idx = fscores_macro.index(opt_fscore_max)
    opt_threshold = thresholds[best_threshold_idx]
    opt_fscore_0 = fscores_0[best_threshold_idx]
    opt_fscore_1 = fscores_1[best_threshold_idx]

    if print_vals==True:
        print('Optimal Result at Threshold = ',opt_threshold)
```

```
        print('F Score for Class 0 = ',opt_fscore_0)
        print('F Score for Class 1 = ',opt_fscore_1)
        print('Non-weighted Mean F Score = ',opt_fscore_max)

    return opt_fscore_max, opt_threshold
```

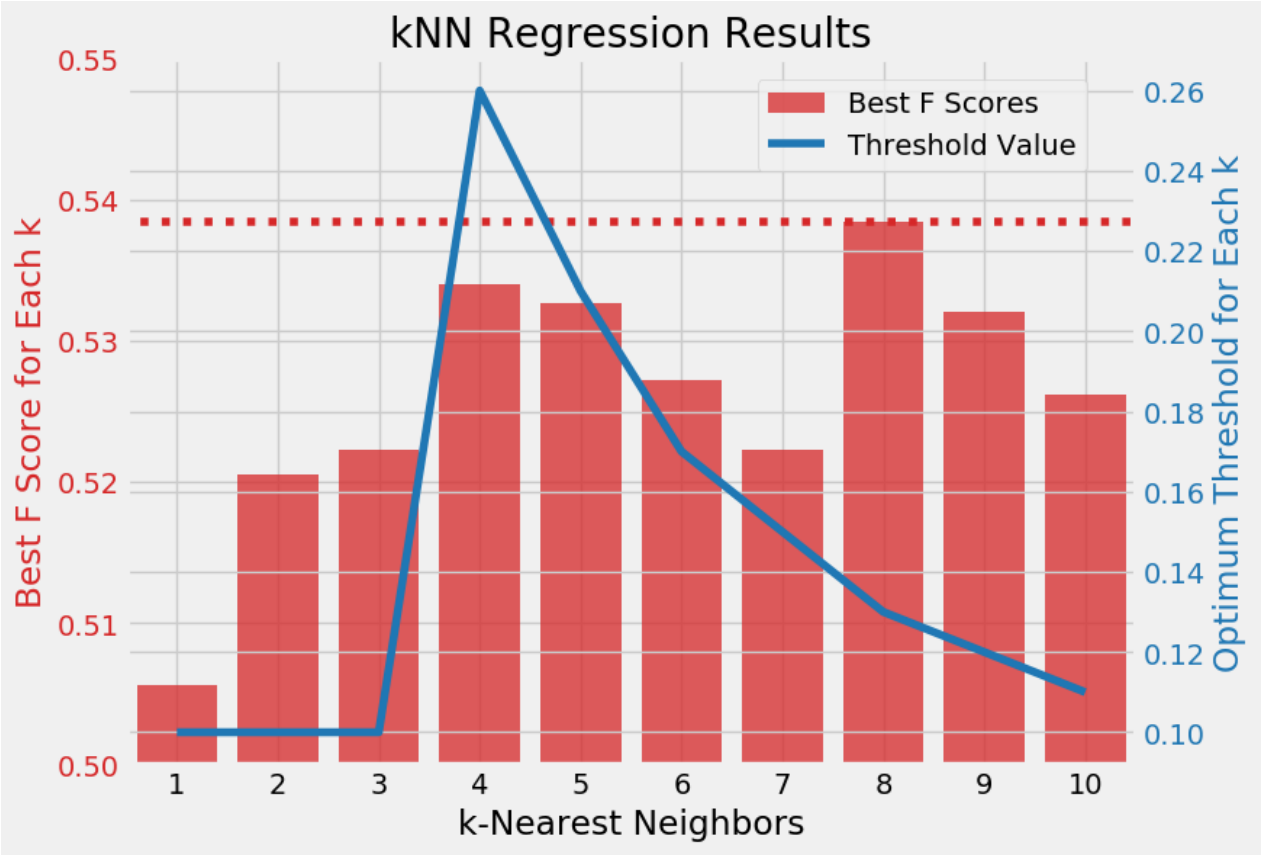
1. k-Nearest Neighbors

Because I am trying to solve a classification problem, kNN Regression could be a good starting point. Points most similar to households that own EVs would be predicted to also own EVs. The model was created using the scikit-learn library. Because the model is sensitive to both the value of k and the value of the threshold, I used values of k from 1 to 10, computing for the optimum threshold and F Score for each k, and storing them each time.

In [609]:
```
from sklearn.neighbors import KNeighborsClassifier

# Set values for k
kneighbors = np.arange(1,11)
thresholds_knn = np.linspace(0.1,0.5,41)
fscores = []
thresholds = []
for k in kneighbors:
    knn_model = KNeighborsClassifier(n_neighbors=k)
    knn_model.fit(X_train,y_train)

    # Obtain probabilities, P(y=0) and P(y=1), for the validation set
    y_prob = knn_model.predict_proba(X_val)
    threshold_vals = determine_threshold(y_prob[:,1],thresholds_knn)
    opt_fscore_max, opt_threshold = determine_optimum_threshold(threshold_vals,print_vals=False
)
    fscores.append(opt_fscore_max)
    thresholds.append(opt_threshold)
```

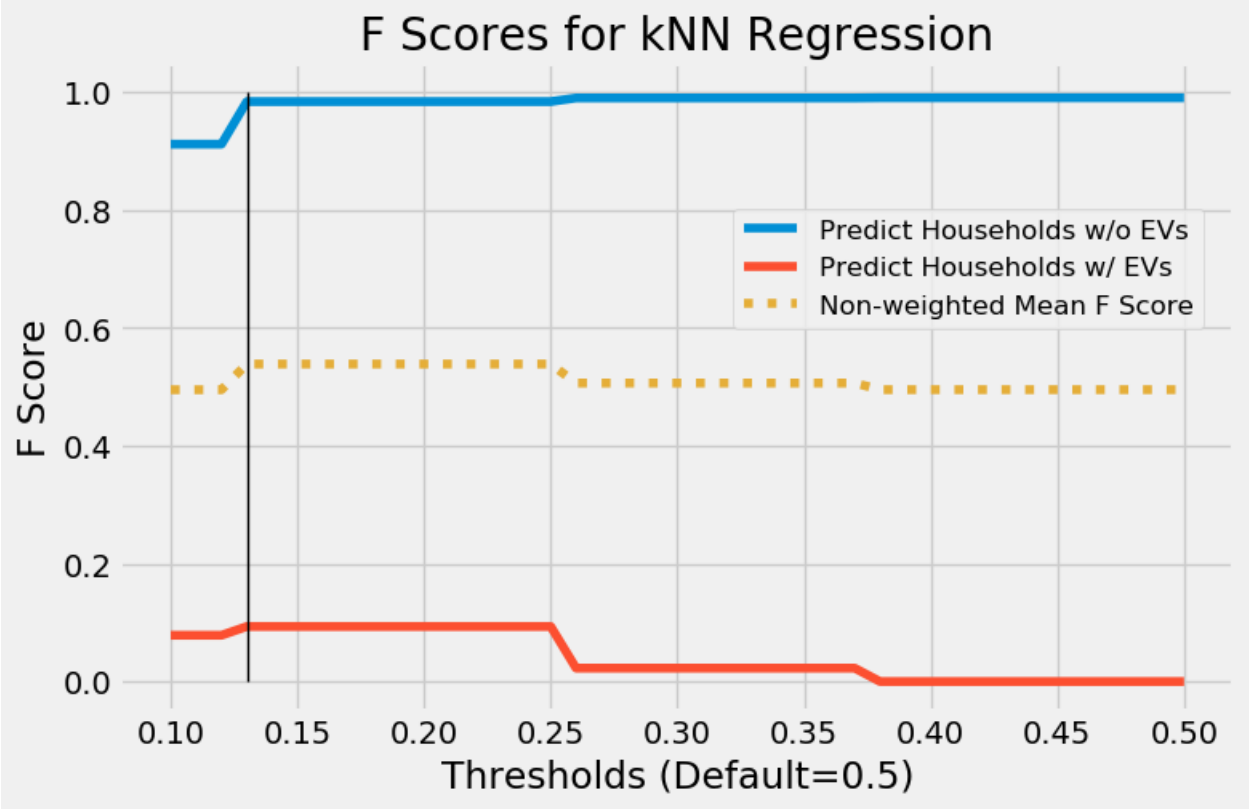
Next, some visualization of the results of the process that just happened above.

In [626]:
```
idx = np.arange(1,len(kneighbors)+1)

fig, ax1 = plt.subplots(figsize=(8,6))

color = 'tab:red'
ax1.set_xlabel('k-Nearest Neighbors')
ax1.set_ylabel('Best F Score for Each k', color=color)
ax1.bar(idx,fscores,label='Best F Scores',color=color,alpha=0.75)
ax1.hlines(max(fscores),0,10.5,color=color,linestyles='dotted')
ax1.tick_params(axis='y', labelcolor=color)
ax1.set_ylim([0.5,0.55])

ax2 = ax1.twinx()

color = 'tab:blue'
ax2.set_ylabel('Optimum Threshold for Each k', color=color)
ax2.plot(idx,thresholds,label='Threshold Value',color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.title('kNN Regression Results')
plt.xticks(np.arange(1,11))
plt.xlim(0.5,10.5)
fig.legend(loc=[0.6,0.8])
plt.show()
```



In the chart above, there are 2 sets of information being shown, the best F score in red and its corresponding optimum threshold in blue for each value of k. We can see here that when k=4, the optimum threshold is at around 0.26, then drops after that. For the most part, the F Scores for each value of k are not too different, but k=8 has the highest F Score. Below, I fitted the model again with this value of k. I then plotted the F Scores for varying thresholds at k=8.

In [423]:
```
max_fscore_idx = np.argmax(fscores)
k_best = kneighbors[max_fscore_idx]

# Fit the final model with best k
knn_model = KNeighborsClassifier(n_neighbors=k_best)
knn_model.fit(X_train,y_train)

# Obtain probabilities, P(y=0) and P(y=1), for the validation set
y_prob = knn_model.predict_proba(X_val)
threshold_vals = determine_threshold(y_prob[:,1],thresholds_knn)
plot_threshold_fscores(threshold_vals,'kNN Regression')
knn_fscore_max, knn_threshold = determine_optimum_threshold(threshold_vals)
```

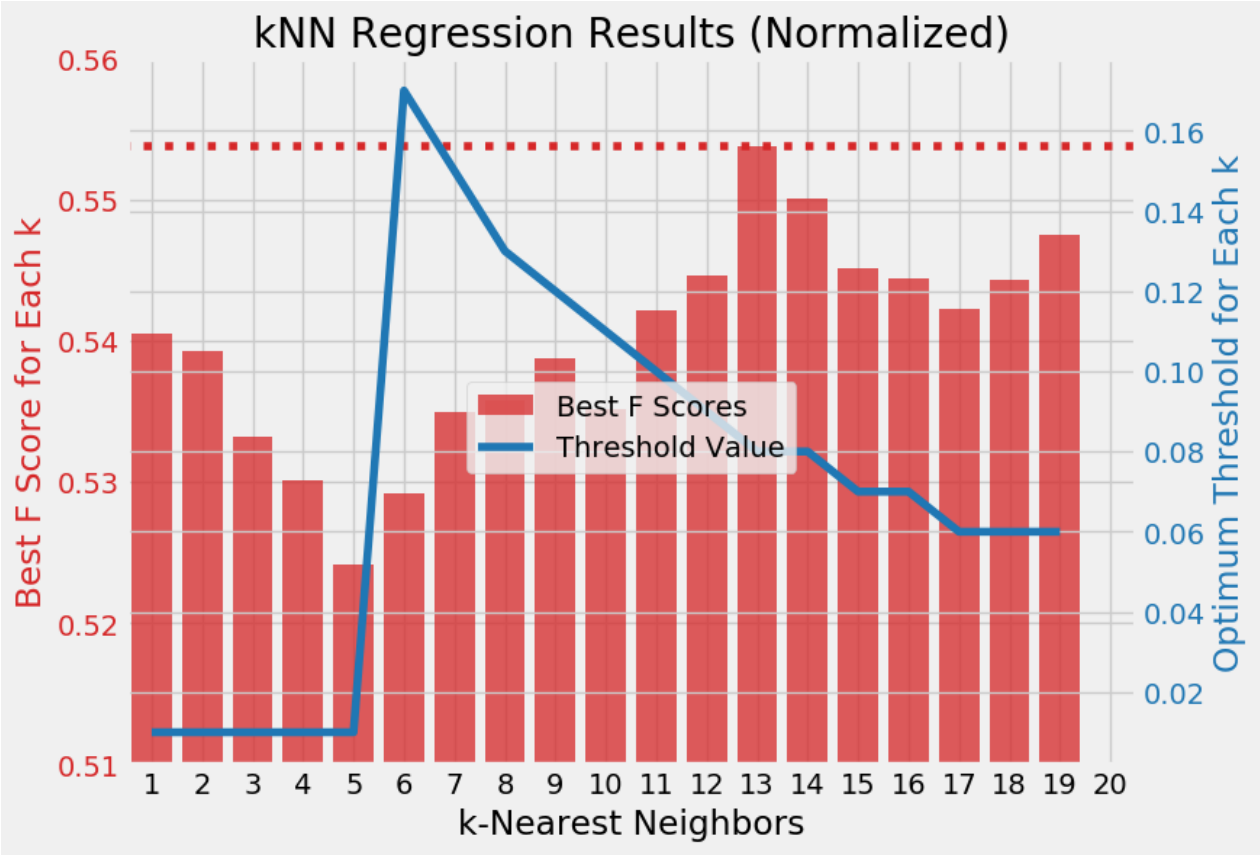



```
Optimal Result at Threshold =  0.13
F Score for Class 0 =  0.9835470602467942
F Score for Class 1 =  0.09333333333333332
Non-weighted Mean F Score =  0.5384401967900637
```

The non-weighted mean F Score of the final model is 0.54, which is generally quite low. Separately, the F Score for Class 1 or EV-owning households is 0.09. Let us see if we can improve this with the normalized set of data, performing the same tasks as I did earlier.

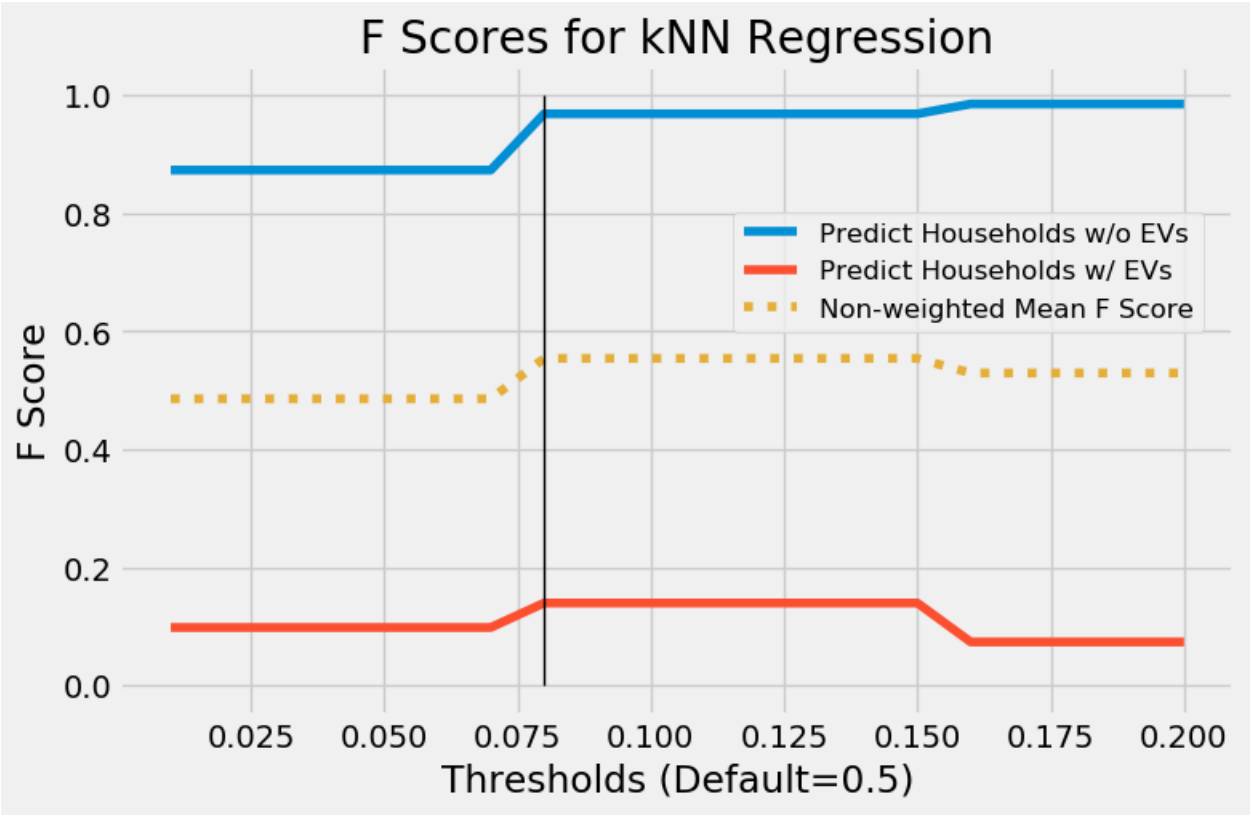
In [638]:
```
thresholds_knn = np.linspace(0.01,0.2,20)
kneighbors = np.arange(1,20)
fscores_norm = []
thresholds_norm = []
for k in kneighbors:
    knn_model_norm = KNeighborsClassifier(n_neighbors=k)
    knn_model_norm.fit(X_train_norm,y_train)

    # Obtain probabilities, P(y=0) and P(y=1), for the validation set
    y_prob_norm = knn_model_norm.predict_proba(X_val_norm)
    threshold_vals = determine_threshold(y_prob_norm[:,1],thresholds_knn)
    opt_fscore_max, opt_threshold = determine_optimum_threshold(threshold_vals,print_vals=False
)
    fscores_norm.append(opt_fscore_max)
    thresholds_norm.append(opt_threshold)
```

In [639]:
```
idx = np.arange(1,len(kneighbors)+1)

fig, ax1 = plt.subplots(figsize=(8,6))

color = 'tab:red'
ax1.set_xlabel('k-Nearest Neighbors')
ax1.set_ylabel('Best F Score for Each k', color=color)
ax1.bar(idx,fscores_norm,label='Best F Scores',color=color,alpha=0.75)
ax1.hlines(max(fscores_norm),0,20.5,color=color,linestyles='dotted')
ax1.tick_params(axis='y', labelcolor=color)
ax1.set_ylim([0.51,0.56])

ax2 = ax1.twinx()

color = 'tab:blue'
ax2.set_ylabel('Optimum Threshold for Each k', color=color)
ax2.plot(idx,thresholds_norm,label='Threshold Value',color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.title('kNN Regression Results (Normalized)')
plt.xticks(np.arange(1,21))
plt.xlim(0.5,20.5)
fig.legend(loc='center')
plt.show()
```



This time, we can observe that when k=6, the optimum threshold surges then drops after that. For the most part, the F Scores for each value of k are not too different, but k=13 has the highest F Score. Below, I fitted the model again with this value of k. I then plotted the F Scores for varying thresholds at k=13.

In [640]:
```
max_fscore_norm_idx = np.argmax(fscores_norm)
k_best_norm = kneighbors[max_fscore_norm_idx]

# Fit the final model with best k
knn_model_norm = KNeighborsClassifier(n_neighbors=k_best_norm)
knn_model_norm.fit(X_train_norm,y_train)

# Obtain probabilities, P(y=0) and P(y=1), for the validation set
y_prob_norm = knn_model_norm.predict_proba(X_val_norm)
threshold_vals = determine_threshold(y_prob_norm[:,1],thresholds_knn)
plot_threshold_fscores(threshold_vals,'kNN Regression')
knn_fscore_max_norm, knn_threshold_norm = determine_optimum_threshold(threshold_vals)
```




```
Optimal Result at Threshold =  0.08
F Score for Class 0 =  0.9680837954405422
F Score for Class 1 =  0.13953488372093023
Non-weighted Mean F Score =  0.5538093395807362
```

The non-weighted mean F Score of the final model is 0.55, which is still relatively low, but is an improvement from the original model. Separately, the F Score for Class 1 or EV-owning households is 0.14. Now I try to see if other models can make better predictions.

2. Regression Trees / Random Forests

Regression Tress and Random Forest are suited for classification problems. Instead of trying to predict a precise number, the nodes branch out to improve the split between two classes. Because the algorithm is based on partitioning off decision boundaries, standardization is also not an issue here.

For easy interpretability, I started with a single regression tree. In this model, I set the random_state to 1 for reproducibility, max_depth to 10 and left the other parameters at their default values. For this exercise, a value for a limiting parameter such as max depth was necessary or else the F Scores would be constant. This is because a very deep tree will provide explicit classifications (0s and 1s), rather than probabilities.

In order to tune the model, I tested different thresholds of probability for the target variable to be predicted as either EV-owning or not.

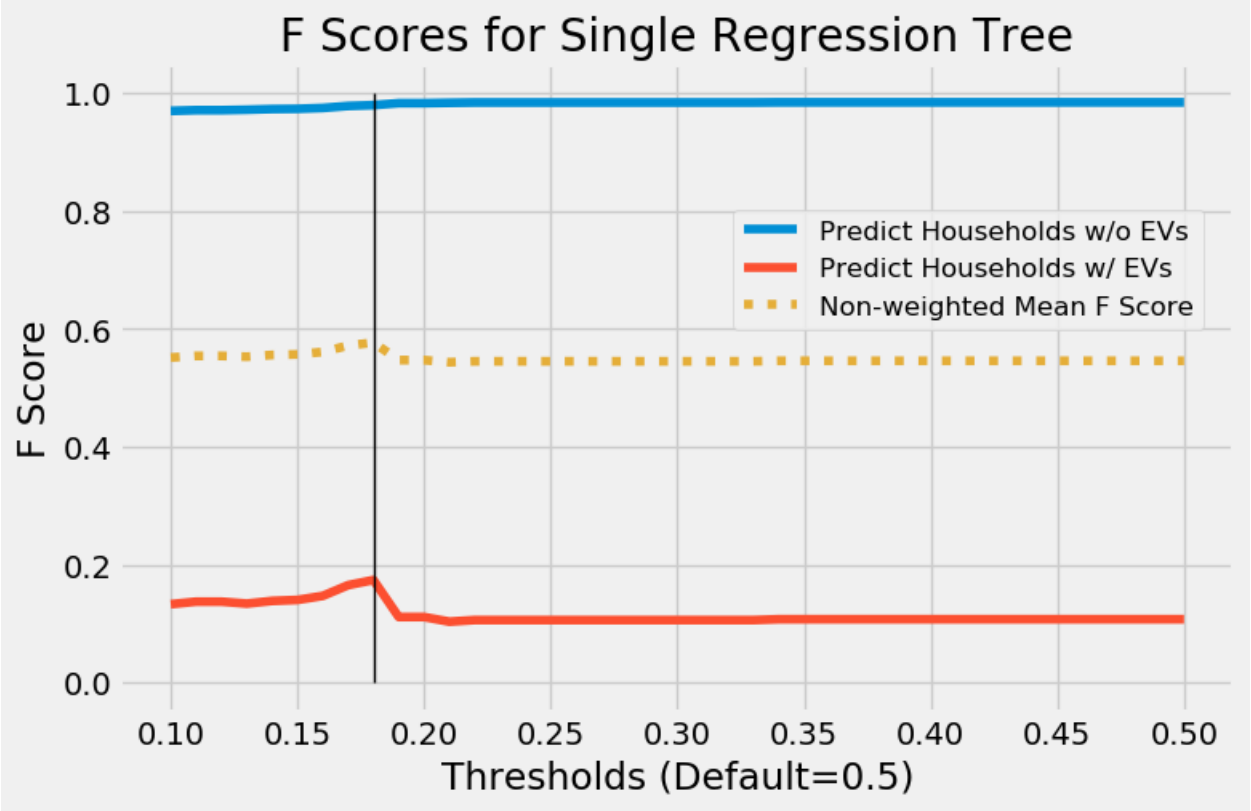
In [424]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
```

In [425]:
```
# Create and fit the model
first_tree = DecisionTreeClassifier(random_state=1,max_depth=10)
first_tree.fit(X_train,y_train)

# Print basic characteristics of regression tree
print("Number of features: {}".format(first_tree.tree_.n_features))
print("Number of nodes (leaves): {}".format(first_tree.tree_.node_count), "\n")

# Obtain and print standard scores for train and validation sets
train_score = first_tree.score(X_train,y_train)
val_score = first_tree.score(X_val,y_val)
print('Train Score: ', train_score)
print('Validation Score: ', val_score)
```

```
Number of features: 32
Number of nodes (leaves): 495

Train Score:  0.9862134537675303
Validation Score:  0.969106463878327
```

As we can see, the train and test scores are very high at 0.99 and 0.97, but these do not provide much meaning because of the imbalance of classes, where only 2.5% of households own EVs in the original data. Below is a visualization of F Scores for each class and both combined. The averaging method applied was 'macro' or non-weighted to give more weight to errors in predicting EV-owning households.

In [426]:
```
thresholds_tree = np.linspace(0.1,0.5,41)

# Obtain probabilities, P(y=0) and P(y=1), for the validation set
y_prob = first_tree.predict_proba(X_val)

# Determine optimum threshold and F scores
threshold_vals = determine_threshold(y_prob[:,1],thresholds_tree)
plot_threshold_fscores(threshold_vals,'Single Regression Tree')
tree_fscore_max, tree_threshold = determine_optimum_threshold(threshold_vals)
```




```
Optimal Result at Threshold =  0.18
F Score for Class 0 =  0.9791692045316116
F Score for Class 1 =  0.17391304347826086
Non-weighted Mean F Score =  0.5765411240049363
```

Here, we can see that for this regression tree, the optimum threshold is at 18%. In this case, if the probability the household owns an EV is greater than 18%, it will be predicted to own an EV. With this parameter, predictions for Class 0 or non-EV do quite well with an F Score of 0.98. On the other hand, predictions for Class 1 or EV don't do very well at 0.17. Their non-weighted mean F score is at 0.58. Later, we will see if a Random Forest Classifier can improve on this result.

Visualization of the Regression Tree

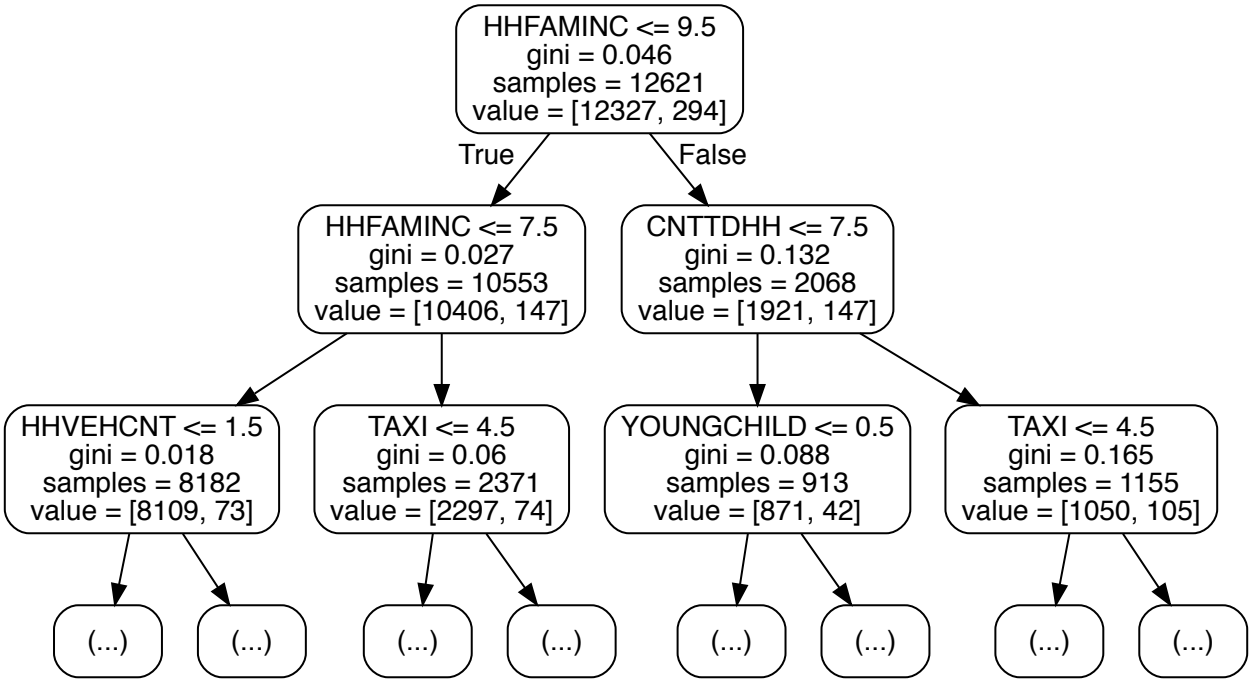
Here's a quick look at the Decision Tree created. Although the model had a max_depth parameter set to 10, the visualization below was limited to a max depth of 2 for easy readability. Very quickly we can see that Household income (HHFAMINC) is an important feature for prediction. Here is a quick dictionary of the features shown below:

HHFAMINC: Household Income
CNTTDHH: Number of Trips Per Day of the Household
HHVEHCNT: Number of Vehicles Owned by the Household
TAXI: Frequency of Taxi Use
YOUNGCHILD: Number of Children Between 0 and 4 Years Old

Furthermore, the gini coefficients are fairly low. On the fourth line of each node, the left value is the number of households without EVs and the right value is the number that do. We can see that nodes that branch to the right more and more tend to have higher ratios of EV-owning to non-EV-owning.

In [427]:
```
import graphviz
from IPython.display import SVG
graph = graphviz.Source(tree.export_graphviz(first_tree, feature_names=X.columns,rounded=True,max_depth=2))
SVG(graph.pipe(format='svg'))
```

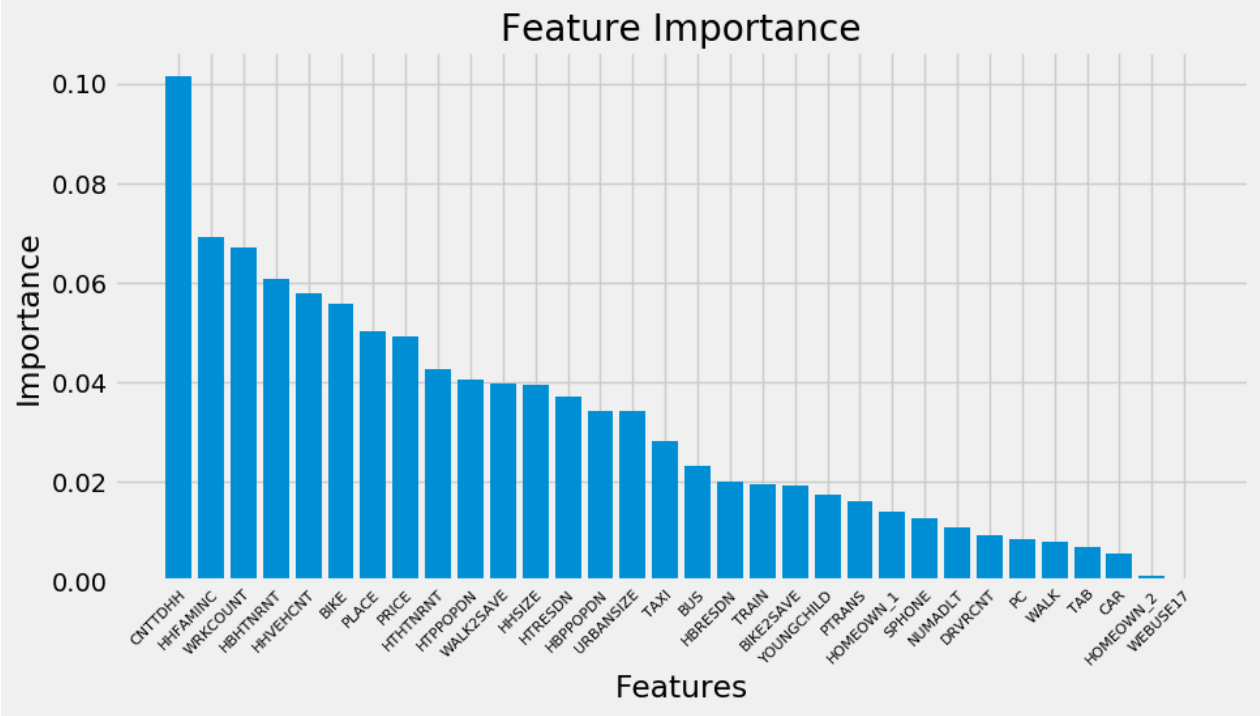
Out[427]:



Visualization of the Feature Importances Based on the Regression Tree

To gain more insights about the Regression Tree created and the relationship of the features with the target variable, I obtained the relative feature importances of each predictor. Below, they are plotted in descending order, where we see that given default parameters for this specific regression tree, the number of trips per day of the household has the most significance.

In [428]:
```
# Obtain Feature Importances and Sort by Importance
feat_impt = pd.DataFrame({'Feature': features.columns,
                          'Importance': first_tree.feature_importances_})
feat_impt = feat_impt.sort_values(by=['Importance'],ascending=False)

# Plot
plt.figure(figsize=(10,5))
plt.bar(feat_impt['Feature'],feat_impt['Importance'])
plt.xticks(rotation=45,rotation_mode='anchor',ha='right',fontsize=8)
plt.title('Feature Importance')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.show()
```



Here, we can see a good spread of feature importance. Based on this regression tree, 'CNTTDHH' or the number of trips per day of the household is the most important feature, followed by the household income, and by the number of working persons in the household.

Random Forest

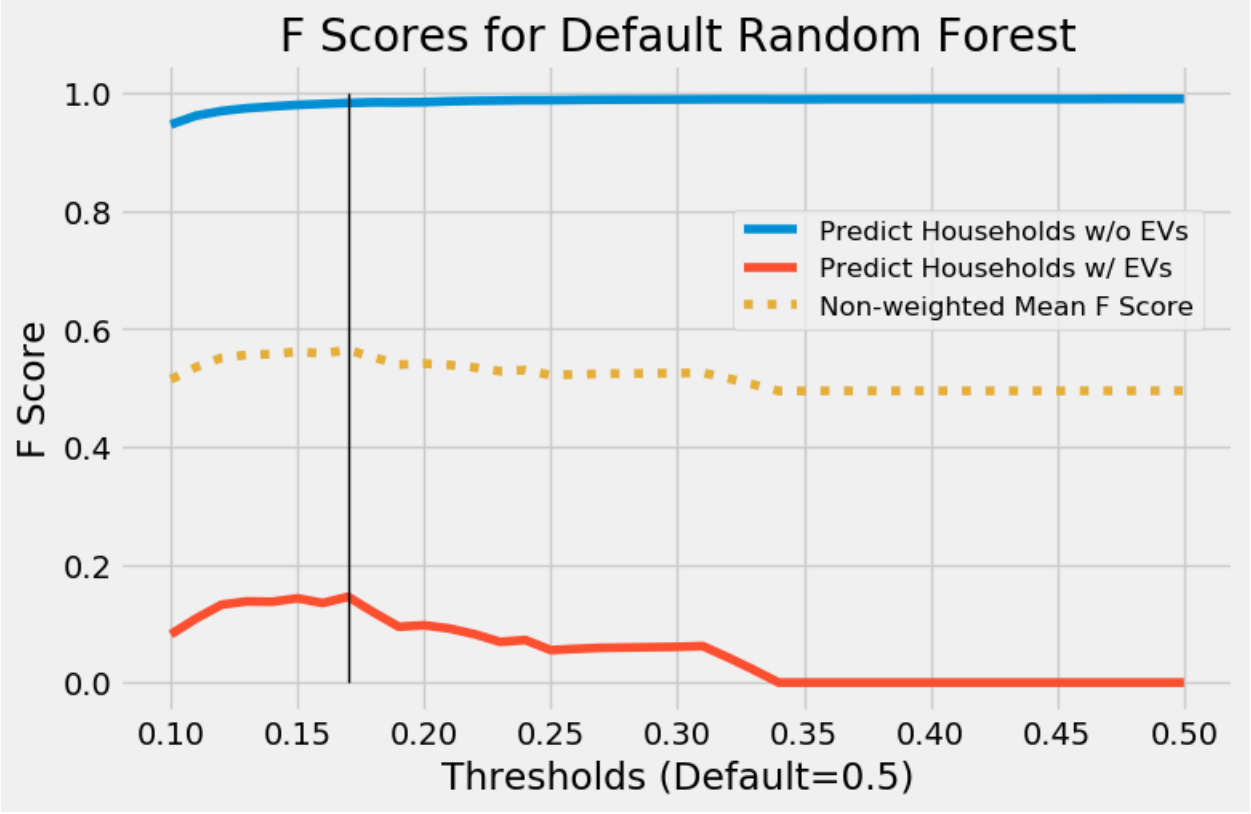
To improve on the methodology and result of the Regression Tree, I implemented a Random Forest model. Below, I first created the model and obtained scores under mostly default parameters. To keep it relatively comparable to the first tree, I set random_state to 1 and max_depth to 10.

In [429]:
```
from sklearn.ensemble import RandomForestClassifier

rf_tree = RandomForestClassifier(random_state=1,max_depth=10)
rf_tree.fit(X_train,y_train)

rf_train_score = rf_tree.score(X_train,y_train)
rf_val_score = rf_tree.score(X_val,y_val)

print('Train Score: ', rf_train_score)
print('Validation Score: ', rf_val_score)
```

```
Train Score:  0.9791617146026463
Validation Score:  0.9798003802281369
```

Again, the scores obtained are high. However, I investigated further by obtaining their F Scores at optimal thresholds as previously done.

In [430]:
```
# Obtain probabilities, P(y=0) and P(y=1), for the validation set
y_prob = rf_tree.predict_proba(X_val)

# Determine optimum threshold and F scores
threshold_vals = determine_threshold(y_prob[:,1],thresholds_tree)
plot_threshold_fscores(threshold_vals,'Default Random Forest')
rf_fscore_max, rf_threshold = determine_optimum_threshold(threshold_vals)
```




```
Optimal Result at Threshold =  0.17
F Score for Class 0 =  0.9829111622833595
F Score for Class 1 =  0.14545454545454545
Non-weighted Mean F Score =  0.5641828538689525
```

Under mostly default parameters, the mean F Score obtained here performs worse than the first tree created earlier. However, I implemented scikit-learn's RandomizedSearchCV to attain the optimal parameters for the Random Forest model. Below, I set uniformly distributed random values for different parameters of RandomForestClassifier. The ranges of distribution are hard-coded and can be subjective, but are ball park figures of what seems both acceptable and computationally reasonable.

For the cross-validation process, I specified 10 folds and 50 iterations or 50 sets of randomized parameters.

In [431]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {'n_estimators': randint(1,30),                    # Number of trees in the forest
              'max_depth': randint(1,20),                       # Maximum depth of the tree
              'min_samples_split': randint(10, 30),             # Minimum number of samples req
uired to split an internal node
              'min_samples_leaf': randint(10, 30),               # Minimum number of samples re
quired to be at a leaf node
              'max_features': randint(1,len(features.columns)), # Number of features to conside
r when looking for the best split
#               'class_weight': [{0:1,1:10}],
              'random_state': [1]
}

rnd_rf_search = RandomizedSearchCV(rf_tree, param_distributions=param_dist, cv=10, n_iter=50)

rnd_rf_search.fit(X_train,y_train)

print(rnd_rf_search.best_params_)
```

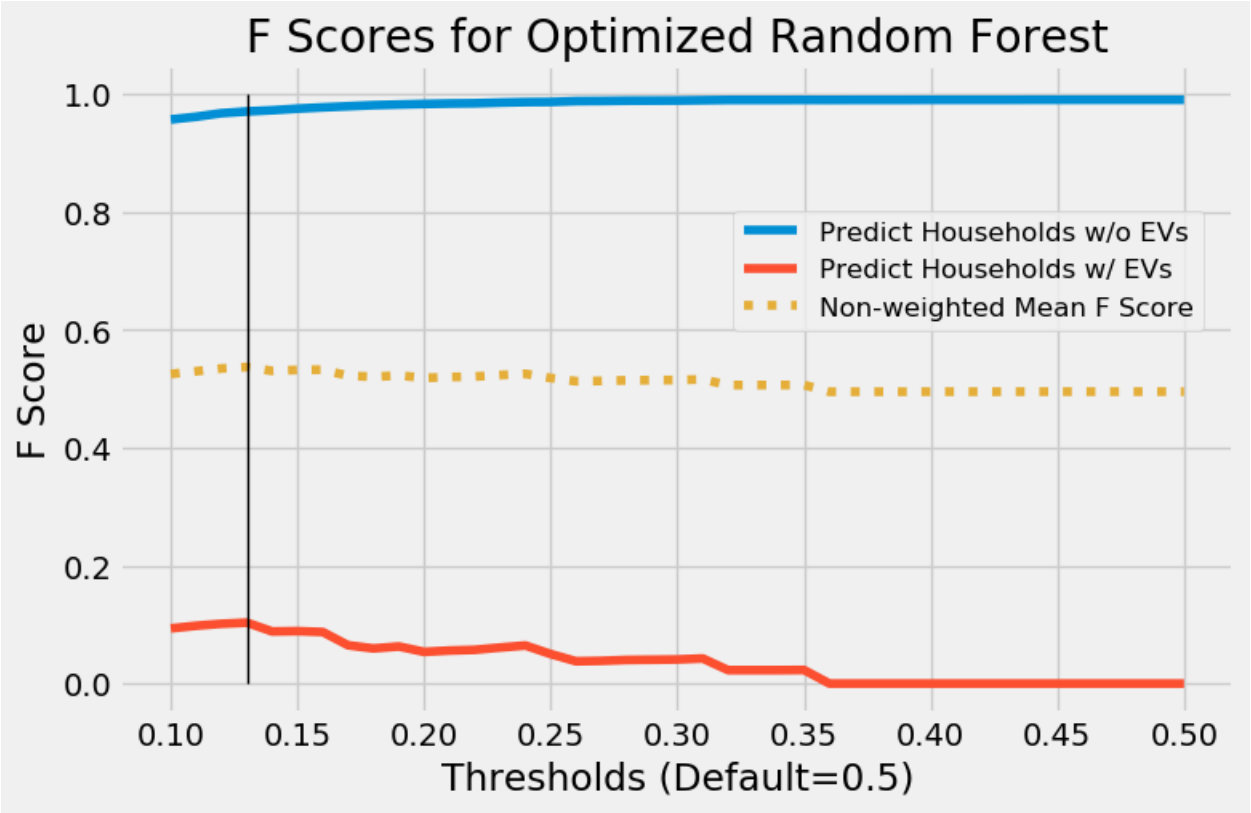
```
{'max_depth': 15, 'max_features': 14, 'min_samples_leaf': 18, 'min_samples_split': 18, 'n_est
imators': 2, 'random_state': 1}
```

After attaining the optimal parameters for the RandomForestClassifier, I then fitted the training data again with these parameters set and obtained their scores.

In [432]:
```
rf_tree.set_params(**rnd_rf_search.best_params_)
rf_tree.fit(X_train,y_train)

rf_train_score = rf_tree.score(X_train,y_train)
rf_val_score = rf_tree.score(X_val,y_val)

print('Train Score: ', rf_train_score)
print('Validation Score: ', rf_val_score)
```

```
Train Score:  0.9767847238729103
Validation Score:  0.9802756653992395
```

In [433]:
```
# Obtain probabilities, P(y=0) and P(y=1), for the validation set
y_prob = rf_tree.predict_proba(X_val)

# Determine optimum threshold and F scores
threshold_vals = determine_threshold(y_prob[:,1],thresholds_tree)
plot_threshold_fscores(threshold_vals,'Optimized Random Forest')
rf_fscore_max, rf_threshold = determine_optimum_threshold(threshold_vals)
```



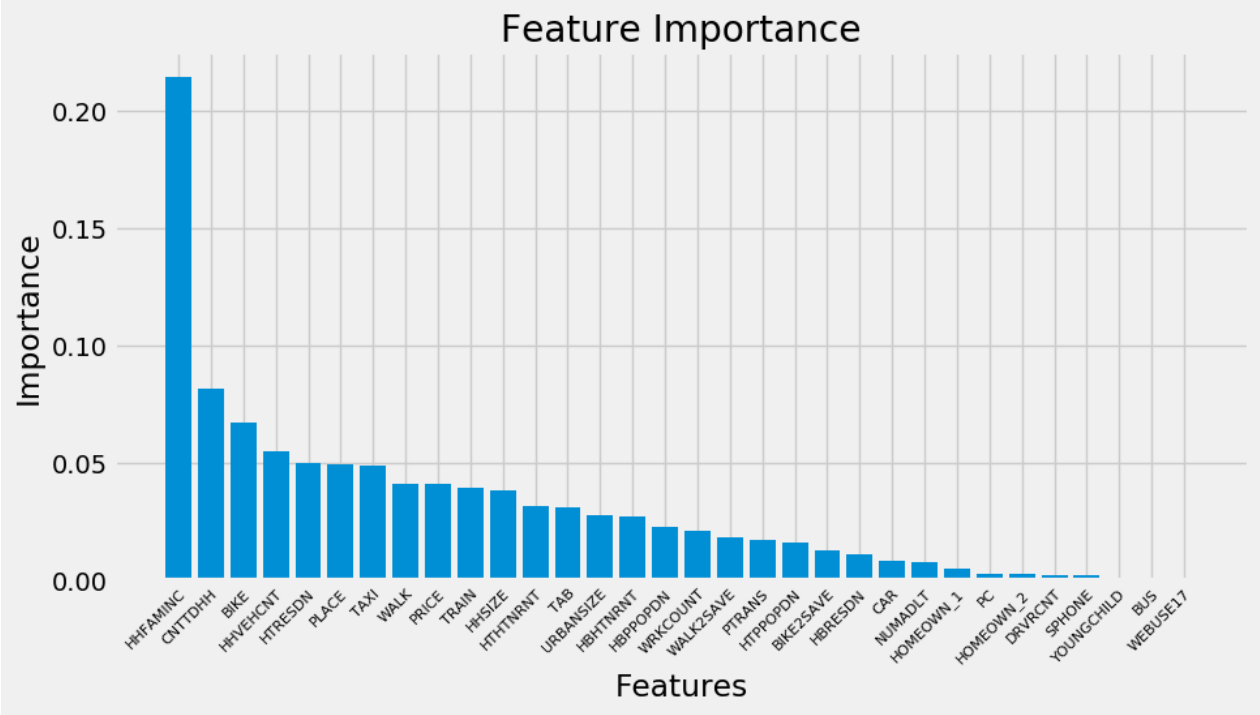

```
Optimal Result at Threshold =  0.13
F Score for Class 0 =  0.9701657458563535
F Score for Class 1 =  0.1033210332103321
Non-weighted Mean F Score =  0.5367433895333428
```

Based on this run, the model marginally improved from the single regression tree. The non-weighted mean F Score improved from 0.56 to 0.58, while the score for Class 1 specifically improved from 0.13 to 0.18.

Visualization of the Feature Importances Based on the Random Forest

Just like earlier, I also obtained and visualized the feature importances based on the random forest model.

In [434]:
```
# Obtain Feature Importances and Sort by Importance
feat_impt = pd.DataFrame({'Feature': features.columns,
                          'Importance': rf_tree.feature_importances_})
feat_impt = feat_impt.sort_values(by=['Importance'],ascending=False)

# Plot
plt.figure(figsize=(10,5))
plt.bar(feat_impt['Feature'],feat_impt['Importance'])
plt.xticks(rotation=45,rotation_mode='anchor',ha='right',fontsize=8)
plt.title('Feature Importance')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.show()
```

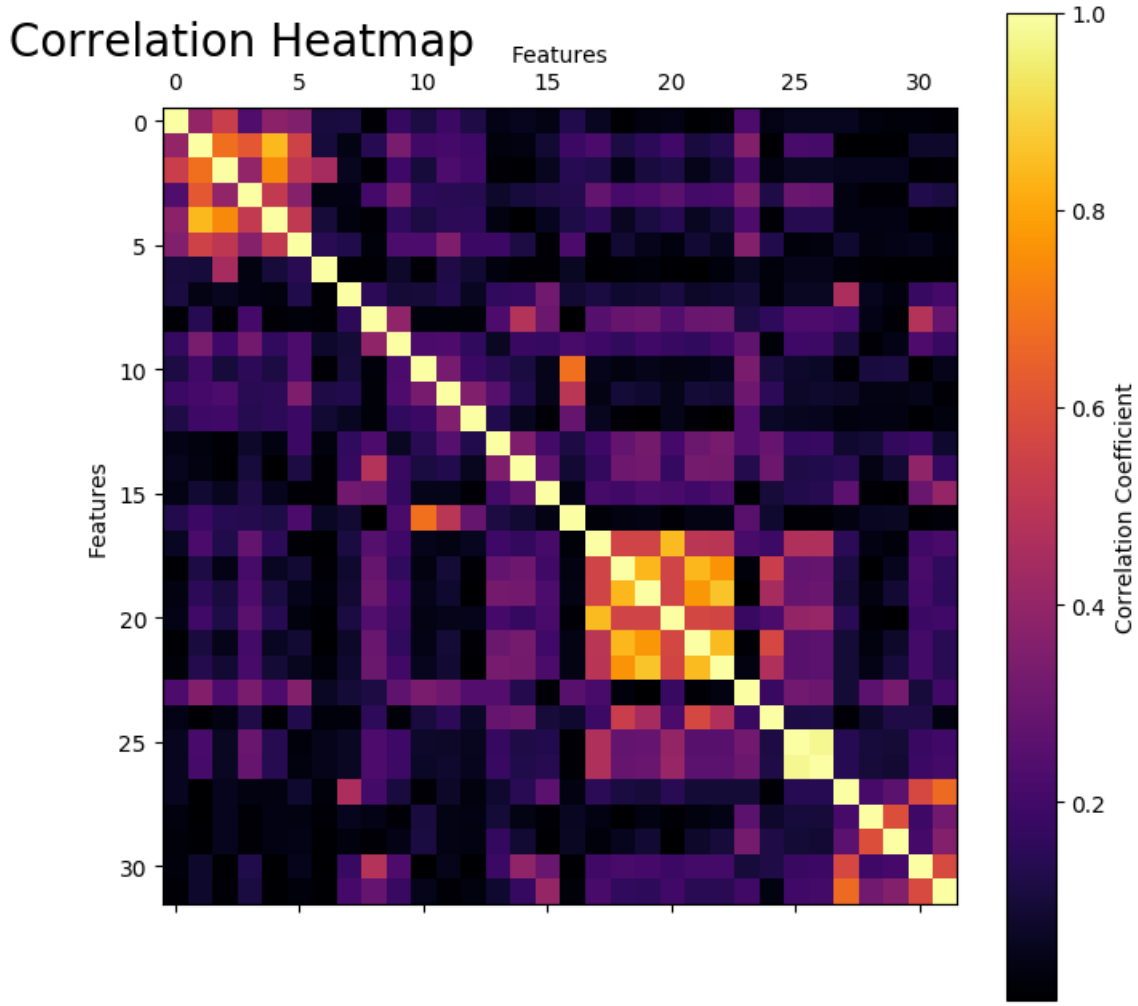


This time, 'HHFAMINC' or household income claims the top spot in feature importance. This was then followed by the previous leader, 'CNTTDHH' or number of trips per day of the household, and 'PLACE' or the household respondent's opinion on travel as a financial burden.

3. Ridge Regression

As a check, I first wanted to determine whether the features are highly correlated to each other. This can help determine the type of regularization to apply. Because there are a lot of pairs of features to inspect, I plotted the correlation matrix on a heatmap shown below. While most of the map shows low correlation, there are also a good amount of high correlation coefficients, disregarding the primary diagonal of course.

In [435]:
```
plt.style.use('default')

fig, ax = plt.subplots(1,1,figsize=(8,8))
ax.xaxis.set_tick_params(labeltop='on')
ax.xaxis.set_tick_params(labelbottom='off')

colormap = ax.imshow(np.abs(features.corr()), cmap='inferno')
ax.set_title('Correlation Heatmap', y=1.05, x=0.10, fontsize=20)
ax.set_ylabel('Features')
ax.set_xlabel('Features')
ax.xaxis.set_label_position('top')

cb = fig.colorbar(colormap)
cb.set_label('Correlation Coefficient')
plt.show()

plt.style.use('fivethirtyeight')
```



Furthermore, although there are a good number of features, they are not so many that I would require coefficients driven to zero for less relevant variables. As such, I've selected the Ridge regularization term to constrain the least squares regression. For this, I will use scikit-learn's Ridge and RidgeCV, as it already incorporates the cross-validation process to tune the model's alpha parameter. For potential values of alpha, I obtained 1000 numbers between 0.0001 and 100.

To start, I first build a model without normalizing the features. Like before, I then determine the threshold for values of the predictions in which a household will be predicted to own an electric vehicle.

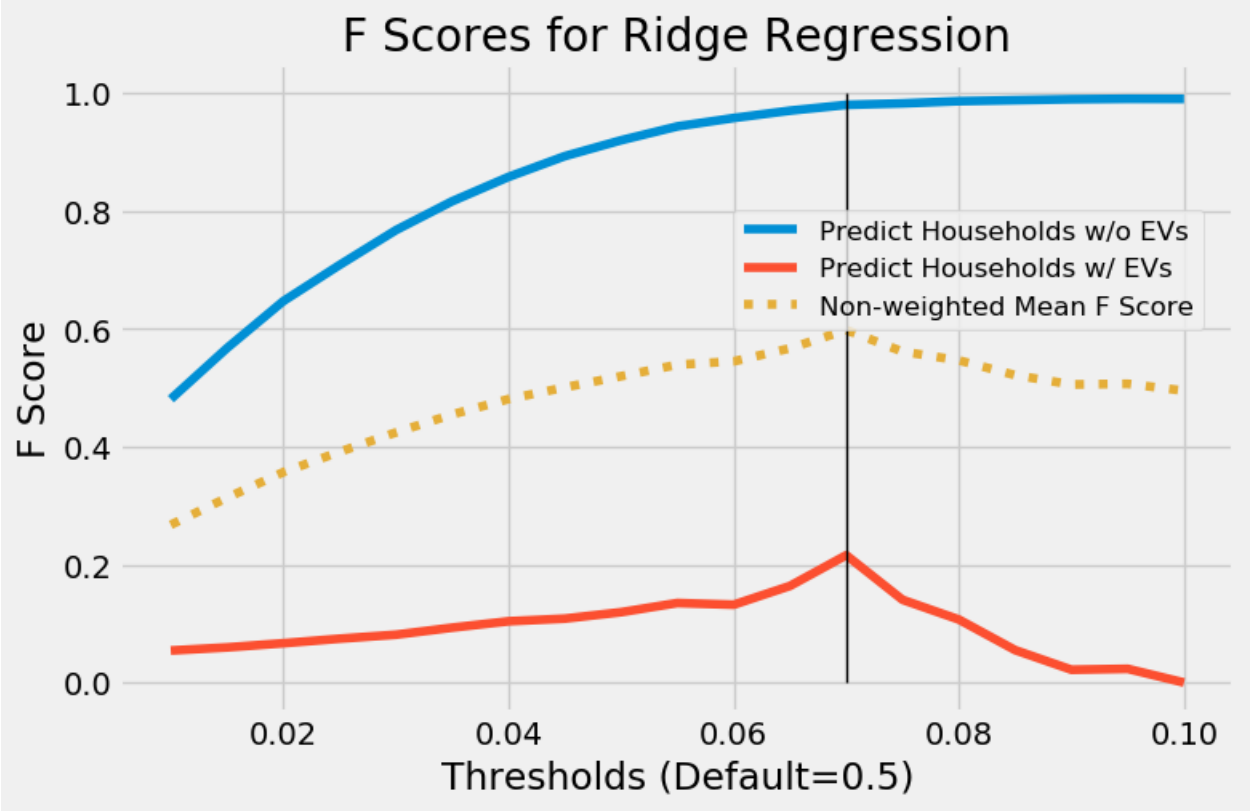
In [436]:
```
from sklearn.linear_model import Ridge,RidgeCV
```

In [437]:
```
thresholds_ridge = np.linspace(0.01,0.1,19)
cv_alphas = np.linspace(0.001, 10000, 1000)

rcv_model = RidgeCV(alphas=cv_alphas,normalize=False)
rcv_model.fit(X_train,y_train)
rcv_alpha = rcv_model.alpha_

print('Alpha:',rcv_alpha)

ridge_model = Ridge(alpha=rcv_alpha,normalize=False)
ridge_model.fit(X_train,y_train)
```

```
Alpha: 1541.5423873873874
```

Out[437]:
```
Ridge(alpha=1541.5423873873874, copy_X=True, fit_intercept=True,
   max_iter=None, normalize=False, random_state=None, solver='auto',
   tol=0.001)
```

In [438]:
```
y_pred = ridge_model.predict(X_val)

# ridge_threshold = determine_threshold(ridge_pred,thresholds_ridge,)
threshold_vals = determine_threshold(y_pred,thresholds_ridge)
plot_threshold_fscores(threshold_vals,'Ridge Regression')
ridge_fscore_max, ridge_threshold = determine_optimum_threshold(threshold_vals)
```




```
Optimal Result at Threshold =  0.07
F Score for Class 0 =  0.9796415945385835
F Score for Class 1 =  0.21596244131455397
Non-weighted Mean F Score =  0.5978020179265687
```

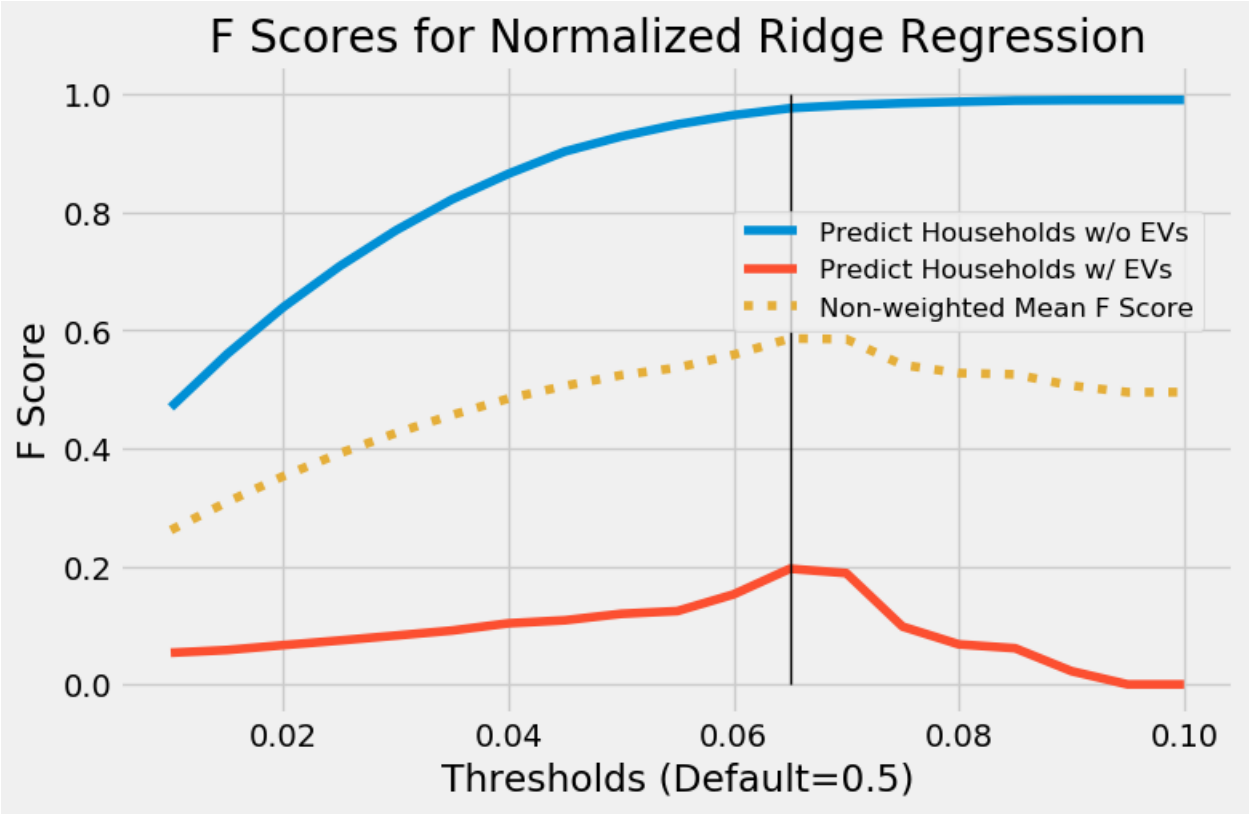
Although the scores are still not very high, this is comparable to and slightly better than the results we obtained in previous models. Next, I normalized the features to try to see if that will improve the model.

In [439]:
```
cv_alphas = np.linspace(0.001, 100, 1000)

rcv_model_n = RidgeCV(alphas=cv_alphas,normalize=True)
rcv_model_n.fit(X_train,y_train)
rcv_alpha_n = rcv_model_n.alpha_

print('Alpha:',rcv_alpha_n)

ridge_model_n = Ridge(alpha=rcv_alpha_n,normalize=True)
ridge_model_n.fit(X_train,y_train)
```

```
Alpha: 0.2011981981981982
```

Out[439]:
```
Ridge(alpha=0.2011981981981982, copy_X=True, fit_intercept=True,
   max_iter=None, normalize=True, random_state=None, solver='auto',
   tol=0.001)
```

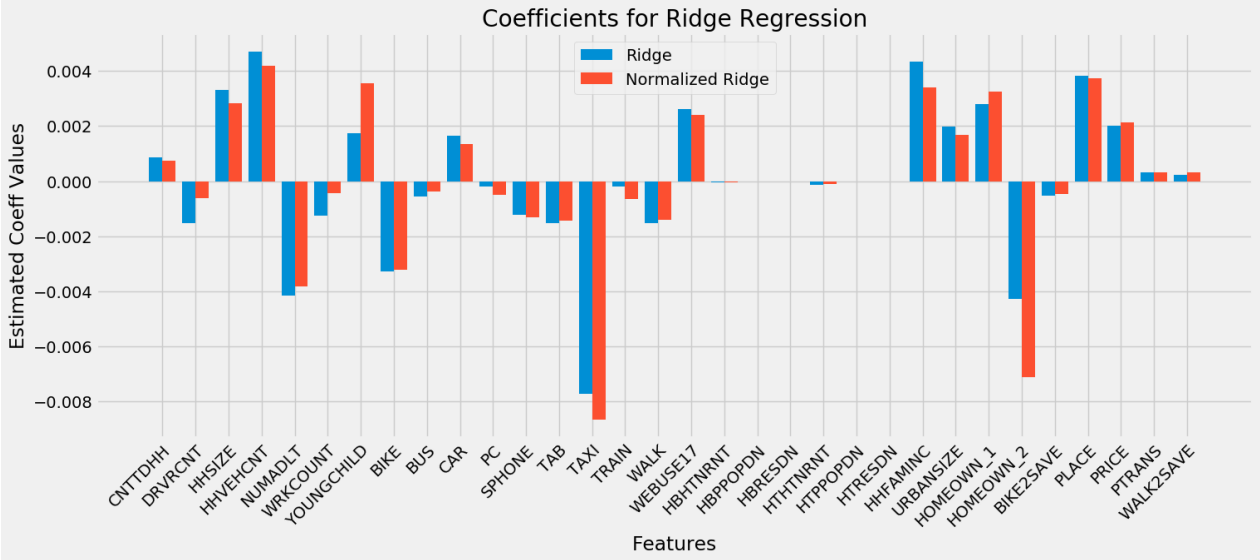
In [440]:
```
y_pred_n = ridge_model_n.predict(X_val)

threshold_vals = determine_threshold(y_pred_n,thresholds_ridge)
plot_threshold_fscores(threshold_vals,'Normalized Ridge Regression')
ridge_fscore_max_n, ridge_threshold_n = determine_optimum_threshold(threshold_vals)
```




```
Optimal Result at Threshold =  0.065
F Score for Class 0 =  0.9758903438991555
F Score for Class 1 =  0.19591836734693877
Non-weighted Mean F Score =  0.5859043556230471
```

The scores above indicate that the normalized model performs marginally worse than the original. However, both are quite close to the values from previous models. To gain some further insight, I plotted the coefficients of the features below.

In [441]:
```
# Numerical X ticks
coeff_idx = np.arange(len(ridge_model.coef_))

plt.figure(figsize=(16,6))
plt.bar(coeff_idx-0.2,ridge_model.coef_,width=0.4,label='Ridge')
plt.bar(coeff_idx+0.2,ridge_model_n.coef_,width=0.4,label='Normalized Ridge')
plt.title('Coefficients for Ridge Regression')
plt.xlabel('Features')
plt.ylabel('Estimated Coeff Values')
plt.xticks(ticks=coeff_idx,labels=features.columns,rotation=45,ha='right',rotation_mode='ancho
r')
plt.legend(loc=9)
plt.show()
```



Viewing this, the features with the highest magnitudes of coefficients are 'TAXI' or frequency of taxi use and 'HOMEOWN_2' or renting of home, which are both negative slopes. Other features with high coefficients include 'HHVEHCNT' or number of vehicles owned by the household, 'HHFAMINC' or household income, and 'PLACE' or opinion on whether travel is a financial burden. While some of these features were also highlighted previously, a couple are more prominent in this model.

Final Test

Now, the models are fitted with the test data that have been left untouched throughout all this time. For this, I constructed a custom function to be able to apply the best thresholds determined earlier. The final scores are then placed as a data frame for easy viewing. Although F Score was the metric used to tune and evaluate the models, it would also be useful to show the precision and recall scores of the results. Remember that for kNN, we are using the normalized model, for Random Forest, we are using the optimized model, and for Ridge, we are using the non-normalized model.

In [442]:
```
from sklearn.metrics import precision_score, recall_score
def model_test(y_prob,threshold):
    y_pred = (y_prob >= threshold).astype(int)
    # Non-weighted Mean F Score
    fscore_macro = f1_score(y_test,y_pred,average='macro')
    # Separate F Scores per Class
    fscore = f1_score(y_test,y_pred,average=None)
    precision = precision_score(y_test,y_pred,average='binary')
    recall = recall_score(y_test,y_pred,average='binary')
    return fscore_macro,fscore[0],fscore[1],precision,recall
```

In [642]:
```
knn_prob = knn_model_norm.predict_proba(X_test_norm)
knn_results = model_test(knn_prob[:,1],knn_threshold_norm)

rf_prob = rf_tree.predict_proba(X_test)
rf_results = model_test(rf_prob[:,1],rf_threshold)

ridge_prob = ridge_model.predict(X_test)
ridge_results = model_test(ridge_prob,ridge_threshold)

pd.DataFrame([knn_results,rf_results,ridge_results],
             index=['k-NN Regression (k=13)','Optimized Random Forest','Ridge Regression'],
             columns=['Macro F Score','Class 0 F Score','Class 1 F Score','Class 1 Precision Sc
ore','Class 1 Recall Score'])
```

Out[642]:

	Macro F Score	Class 0 F Score	Class 1 F Score	Class 1 Precision Score	Class 1 Recall Score
k-NN Regression (k=13)	0.533464	0.961972	0.104956	0.073469	0.183673
Optimized Random Forest	0.545316	0.964127	0.126506	0.089744	0.214286
Ridge Regression	0.571763	0.973642	0.169884	0.136646	0.224490

In general, the scores are not very high. However, these numbers need to be put into context, which will be discussed in the next section. Based on the results, we can see that Ridge Regression performed the best among the three models.

Previously, I mentioned that 2.5% of households in California own EVs. However, that statistic was weighted on the household weight, or how much a household is representative of the population. Since this weight was not applied in the model, let's try to see the percentage when it is unweighted and the percentage specifically for the test set.

In [516]:
```
print('CA Households with EVs (Unweighted):',sum(target)/len(target))
print('CA Households with EVs in Test Set (Unweighted):',sum(y_test)/len(y_test))
```

```
CA Households with EVs (Unweighted): 0.022579265104339972
CA Households with EVs in Test Set (Unweighted): 0.02328897338403042
```

It's similar - unweighted, the percentage is at about 2.3% of households.

Let us attempt to apply this model to another state like Texas, which again is arbitrarily selected. Because it's hard to individually asses the results, we will be looking at the total number of households in Texas predicted to own an EV. Let us first take a look at the current scenario. The following shows the total number of households that own EVs in Texas (after the cleaning process which removed households).

In [507]:
```
print('Current Households with EVs:',sum(target_tx['EV_bin']*target_tx['WTHHFIN']))
```

```
Current Households with EVs: 31980.723516038237
```

In [508]:
```
# Predict using Texas data
ridge_prob_tx = ridge_model.predict(features_tx)
# Use the same threshold
y_pred_tx = (ridge_prob_tx >= ridge_threshold).astype(int)
print('Predicted Households with EVs:',sum(y_pred_tx*target_tx['WTHHFIN']))
```

```
Predicted Households with EVs: 193495.27522579313
```

Interpretation and Conclusions (20 points)

First, we look at the model performance. The scores of the models are all fairly low, with precision ranging from 7 to 14%, and recall ranging from 18 to 22%. What do these numbers mean? For precision, this means that of all the households predicted to own EVs, only 14% of them truly do. For recall, this means that of all the households that own EVs, I was only able to predict 22% of them. While these numbers seem low, let us put it into context. Only 2.3% of households in California own EVs (unweighted). Loosely, if one were to randomly sample 2.3% of households and predict them all to own EVs, it would be 2.3% correct. Hence, the model provided results that are at least six times better than random. Of course, many changes to the model can be made for improvement, such as changing the evaluation metric, applying multifold cross validation on threshold selection (rather than a single split for validation), narrowing the scope such that the proportion of EV owners is higher, or adding other data sources and more features to the model. After all, it is also possible that the characteristics of the data are insufficient to make clear predictions.

The final prediction made in Texas of a six-fold increase in EV ownership should clearly be taken with a grain of salt. The model result can be investigated deeper by analyzing which categories of households were false positives. Resources can be allocated to further study the potential for these identified households to purchase EVs in the near future and how programs can be formulated to facilitate such change. Whether they're in California or in Texas, these false positives seem to have the most in common with current EV owners, making them 'low-hanging fruit'.

Aside from the meager performance of the models, there are also a lot of factors unaccounted for in the model. As warned in the first sections, the models would not account for factors difficult to quantify, such as culture and politics. California would also have a different set of policies and incentives from Texas that could be the cause of higher adoption. Another main caveat is that the data used is merely a snapshot in time. It does not incorporate the rate of adoption, EV sales information, change in price of the vehicles, change in price of fuel, change in environmental policy, and so on.