PROBABILISTIC AND DETERMINISTIC REGRESSION FOR ESTIMATING AVERAGE MONTHLY SPEND

By George Pinto

PROBLEM STATEMENT

A Bike Manufacturing Company wants to know If they can accurately predict how much customers will spend on a monthly basis by creating a model using their existing data

Based on the results of the model their marketing efforts can be directed strategically by (for example) ranking customers according to their monthly expenses on company products and directing their efforts toward customers that are more likely to respond

PROBLEM STATEMENT

- Three datasets are available
- CSV file containing customer information
- CSV file containing average monthly spend information (target)
- CSV file containing information regarding whether they are bike buyers or not

DATA UNDERSTANDING AND CLEANING

It is extremely important to understand what type of solution this problem requires and what algorithm we would use to solve this problem before we deal with the data

Average monthly spend is our target, this is a continuous numerical variable – because we are given the target and we can use it for training this problem is supervised

Because our target is continuous, our algorithm of choice should be a form of regression and our best choice to start is multiple linear regression without regularization since it is best to try to solve the problem with the simplest most explainable format possible

DATA UNDERSTANDING AND CLEANING

- Since we are using Python, and now we know that we have a multiple regression problem we need to choose some libraries for this purpose
- For data exploration, wrangling and data set creation we are choosing pandas
- For model creation we are choosing Sci-Kit Learn

DATA UNDERSTANDING AND CLEANING

- Merge the 3 data sets together using pandas:

```
[ ] # Do inner join on Customer ID, we'll call the dataset AWS for adventure works

    AWS = pd.merge(Customers, AveMonthSpend, on='CustomerID')
    AWS.head()
```

	CustomerID	Title	FirstName	MiddleName	LastName	Suffix	AddressLine1	AddressLine2	City	StateProvinceName	Cou
0	11000	NaN	Jon	V	Yang	NaN	3761 N. 14th St	NaN	Rockhampton	Queensland	
1	11001	NaN	Eugene	L	Huang	NaN	2243 W St	NaN	Seaford	Victoria	

```
# Merge third data set also with an inner join on Customer ID

AWS = pd.merge(AWS, BikeBuyer, on='CustomerID')
AWS.head()
```

ıder	MaritalStatus	HomeOwnerFlag	NumberCarsOwned	NumberChildrenAtHome	TotalChildren	YearlyIncome	AveMonthSpend	BikeBuyer
M	M	1	0	0	2	137947	89	0
M	S	0	1	3	3	101141	117	1

DATA UNDERSTANDING AND CLEANING

- Identify null values and value of feature columns in the context of regression, we only need one identifier for customers (for example), the company can consolidate customer names with the customer id and the id is meaningless in the context of predictor variables for our algorithm:

```
# Let's see which columns have null values and how many nulls

AWS.isnull().sum()
```

```
CustomerID              0
Title               17121
FirstName               0
MiddleName           7189
LastName                0
Suffix              17207
AddressLine1            0
AddressLine2        16918
City                    0
StateProvinceName       0
```

DATA UNDERSTANDING AND CLEANING

- Delete columns that will not add value to the prediction problem:

```
# We know the customer ID column is complete
# These columns have no prediction value and can be safely dropped. The rows will retain
# the information we need for prediction

AWS.drop(columns=['Title','Suffix','AddressLine2','MiddleName'], inplace=True)
```

```
[ ] # Check for nulls again

    AWS.isnull().any()
```

```
CustomerID            False
FirstName             False
LastName              False
AddressLine1          False
City                  False
StateProvinceName     False
```

DATA UNDERSTANDING AND CLEANING

```
# Check the shape

AWS.shape
```

```
(17209, 21)
```

```
# Check the data types after the merge

AWS.dtypes
```

```
CustomerID                 int64
FirstName                 object
LastName                  object
AddressLine1              object
City                      object
StateProvinceName         object
CountryRegionName         object
PostalCode                object
PhoneNumber               object
BirthDate                 object
Education                 object
Occupation                object
Gender                    object
MaritalStatus             object
HomeOwnerFlag              int64
NumberCarsOwned            int64
NumberChildrenAtHome       int64
```

DATA UNDERSTANDING AND CLEANING

- Remove any duplicate rows
- If feature variables are highly correlated to each other, keep those that are more highly correlated to the target variable (have more predictive value):

```
# Which one explains more of the variance of 'AveMonthSpend'

AWS[['NumberChildrenAtHome','TotalChildren','AveMonthSpend']].corr()
```

	NumberChildrenAtHome	TotalChildren	AveMonthSpend
NumberChildrenAtHome	1.000000	0.647287	0.730169
TotalChildren	0.647287	1.000000	0.500206
AveMonthSpend	0.730169	0.500206	1.000000

```
# We'll keep 'NumberChildrenAtHome' as it is more informative and drop the other

AWS.drop('TotalChildren', axis=1, inplace=True)
```

DATA UNDERSTANDING AND CLEANING

- If a feature is presented as a continuous variable but can be useful in segregating customers into groups (for example age groups) consider binning into categories:

```
AWS['Age'].describe()
```

```
count    16471.000000
mean        36.446725
std         11.244468
min         18.000000
25%         28.000000
50%         35.000000
75%         44.000000
max         88.000000
Name: Age, dtype: float64
```

```
# It would be better to bin age so that we can group by Avg gender, Age and Marital Status and
# check their effects on AveMonthSpend

bins = [10, 20, 30, 40, 50, 60, 70, 80, 90]
group_names = ['10-19', '20-29', '30-39', '40-49', '50-59', '60-69', '70-79', '80-89']
AWS['Decade'] = pd.cut(AWS['Age'], bins, labels=group_names)
```

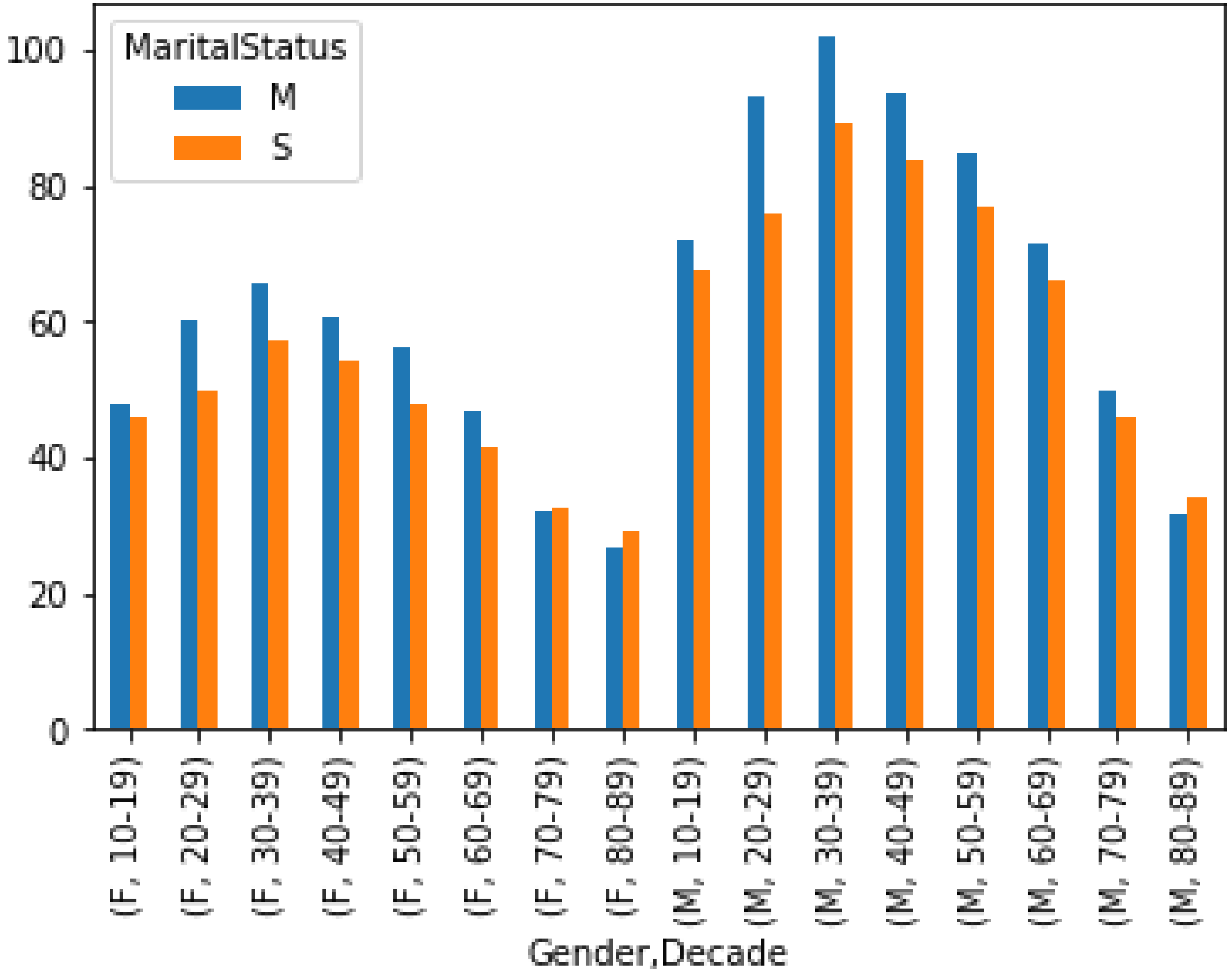
DATA UNDERSTANDING AND CLEANING

- We can use the binned age feature to see how helpful other features are in explaining the variance of the target feature:

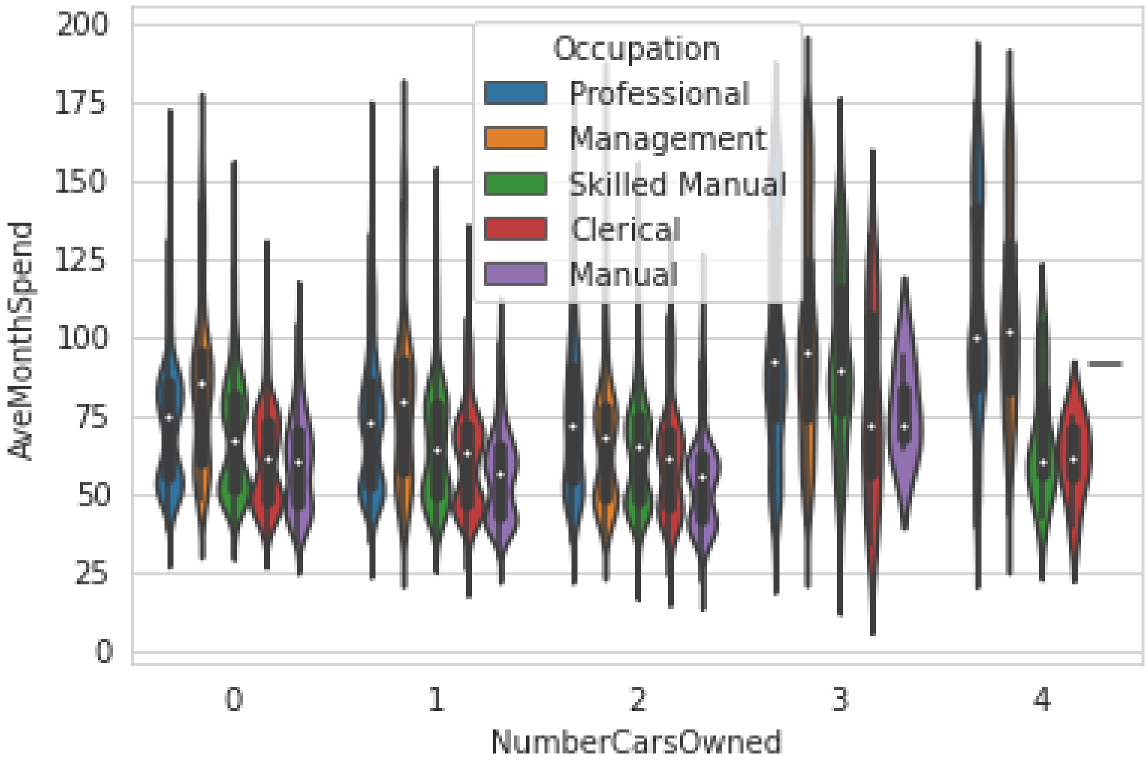
```
HighestSpend = AWS.groupby(['Gender','Decade','MaritalStatus']).mean()
HighestSpend = pd.DataFrame(HighestSpend)
HighestSpendSorted = HighestSpend.sort_values(by='AveMonthSpend')
HighestSpendSorted
```

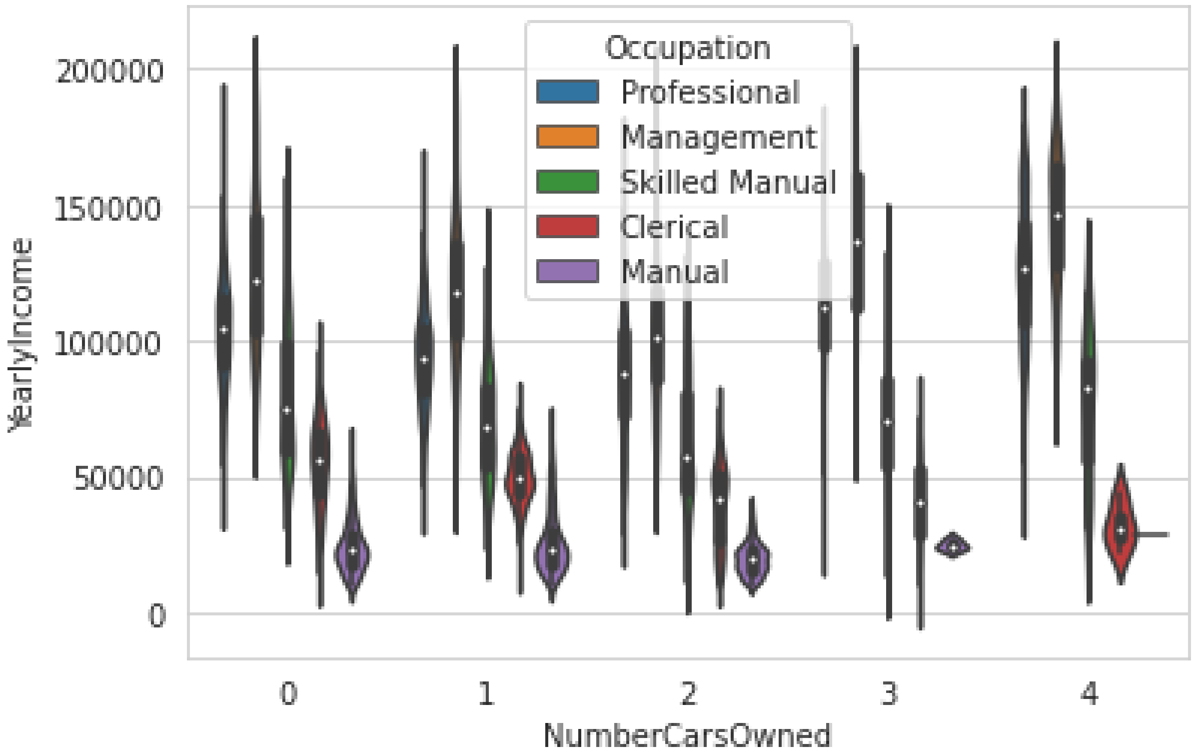
```
# Visualize 'AveMonthSpend' it looks like 'Gender' and 'MaritalStatus' are very helpful in explaining
# the variance

HighestSpendSorted['AveMonthSpend'].unstack().plot(kind='bar');
```

DATA EXPLORATION



DATA EXPLORATION





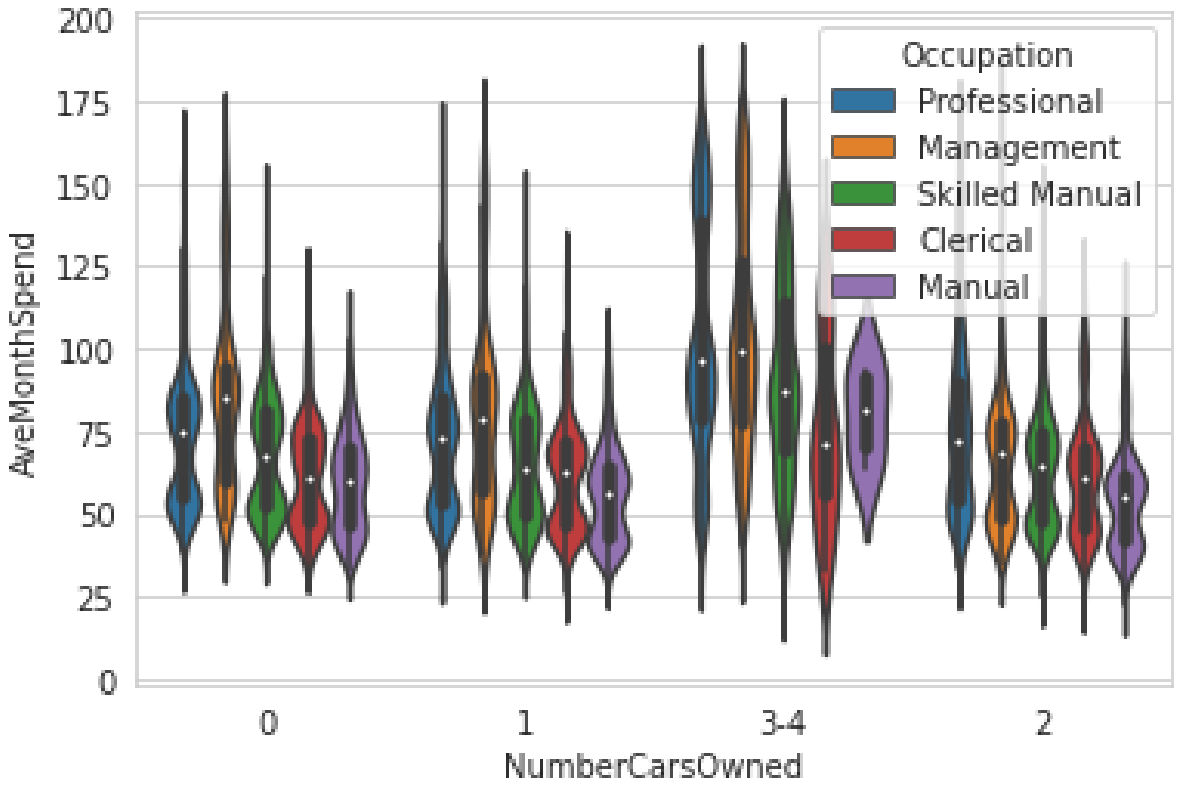
DATA EXPLORATION

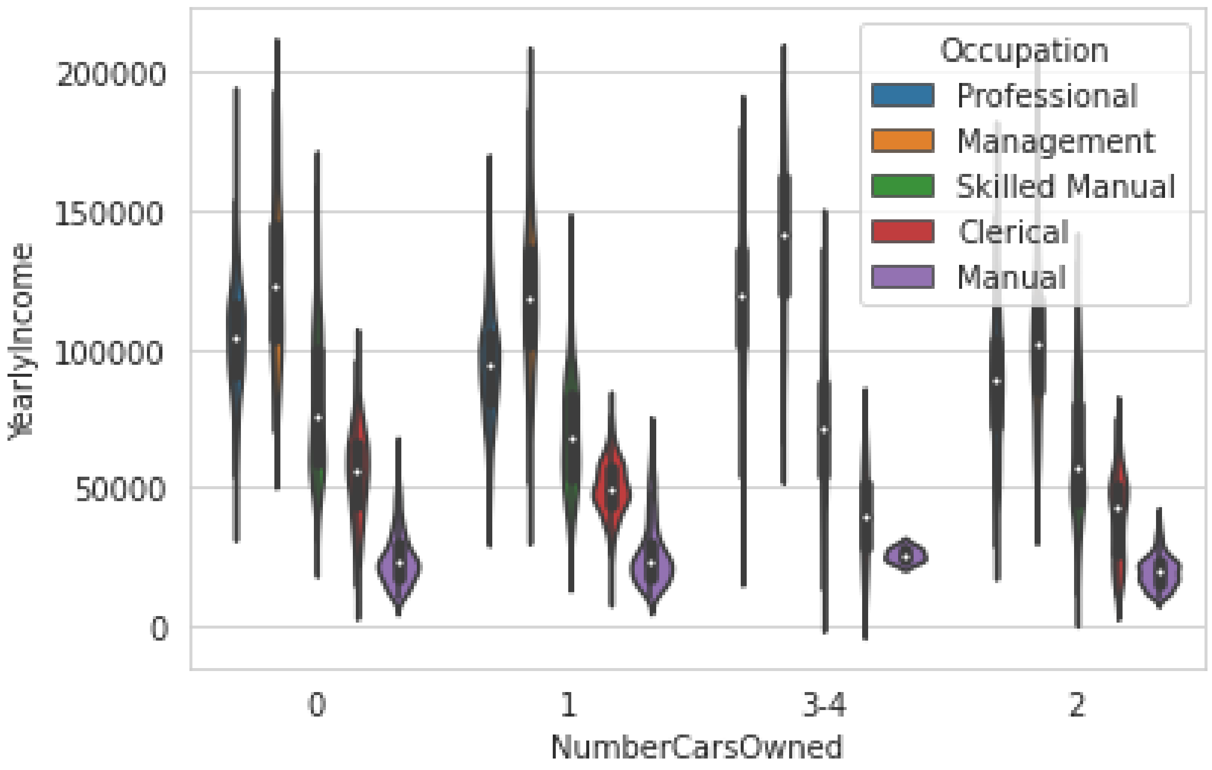
- We need to correct this issue of the flat violin plot (another type of missing data, perhaps manual laborers who replied to the survey data simply did not have that many cars, by aggregating categories the information will become useful to our model, we'll do 3 to 4 cars by combining the 3 and 4 cars owned categories :

```
[ ]  # People that do manual labor just cannot afford 4 cars so we will group the last car group at 3-4

     NumberCarsOwnedCategories = {0:'0', 1:'1',
                                  2:'2', 3:'3-4', 4:'3-4'}
     AWS['NumberCarsOwned'] = [NumberCarsOwnedCategories[x] for x in AWS['NumberCarsOwned']]
     AWS['NumberCarsOwned'].value_counts()
```

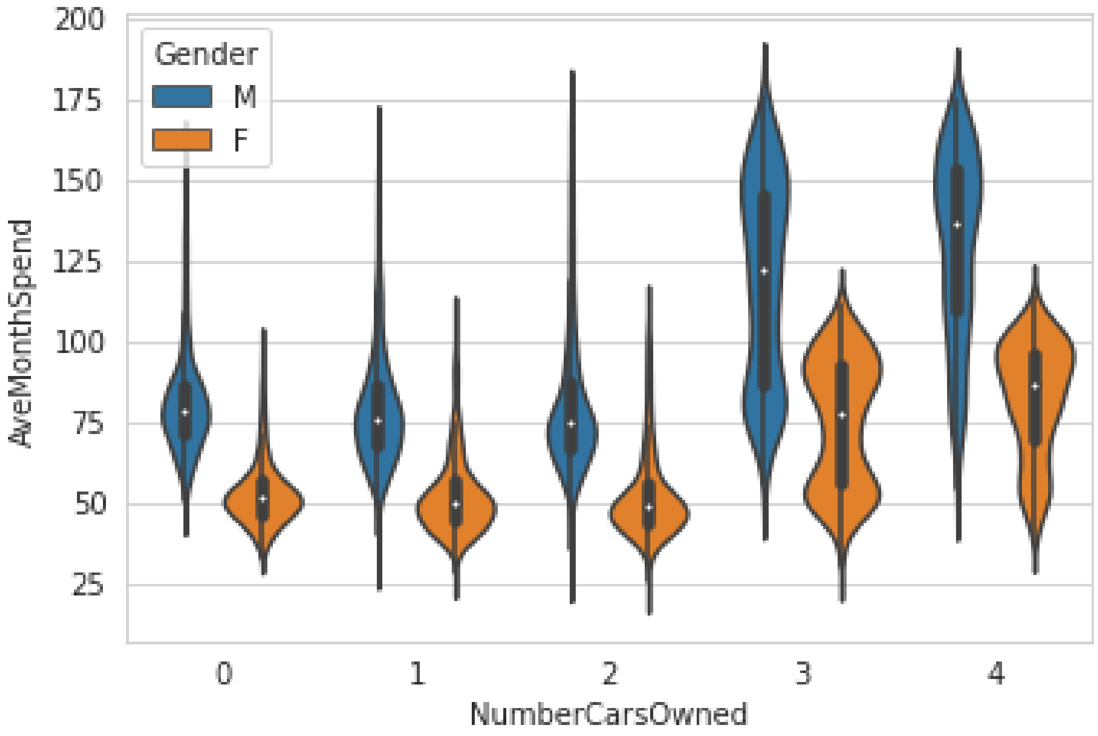
```
2      5767
1      4357
0      3788
3-4    2559
Name: NumberCarsOwned, dtype: int64
```

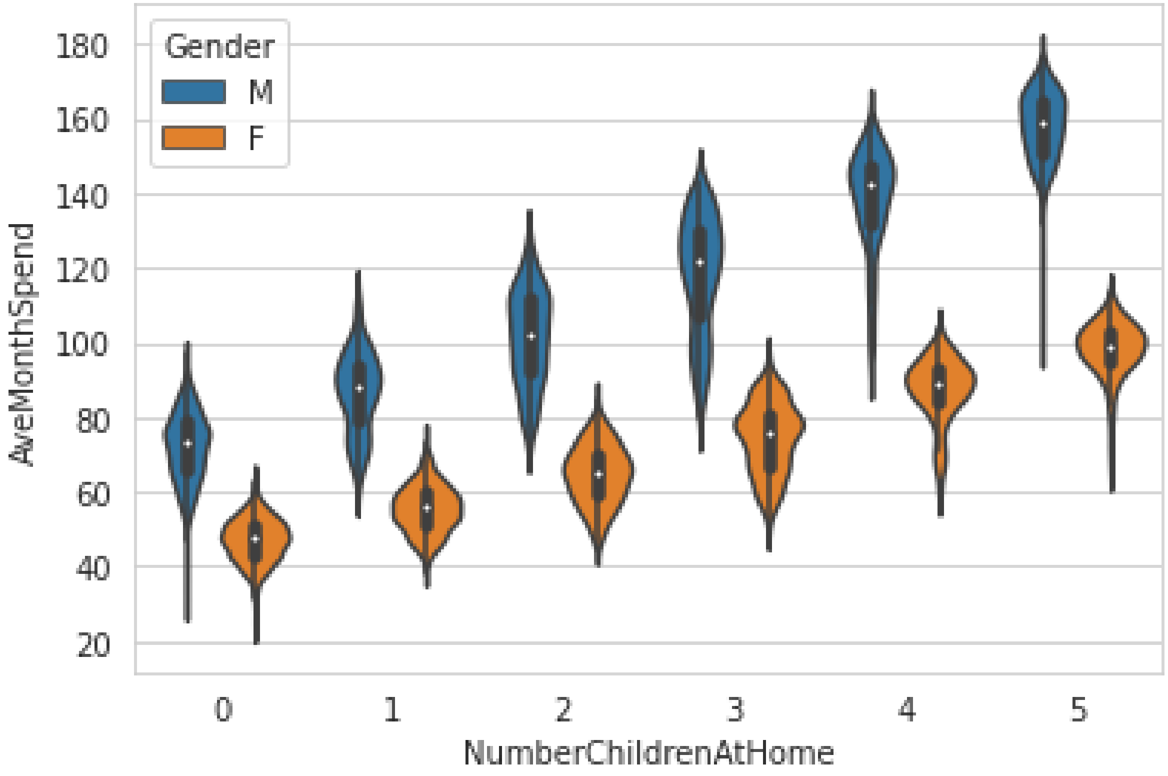
DATA EXPLORATION





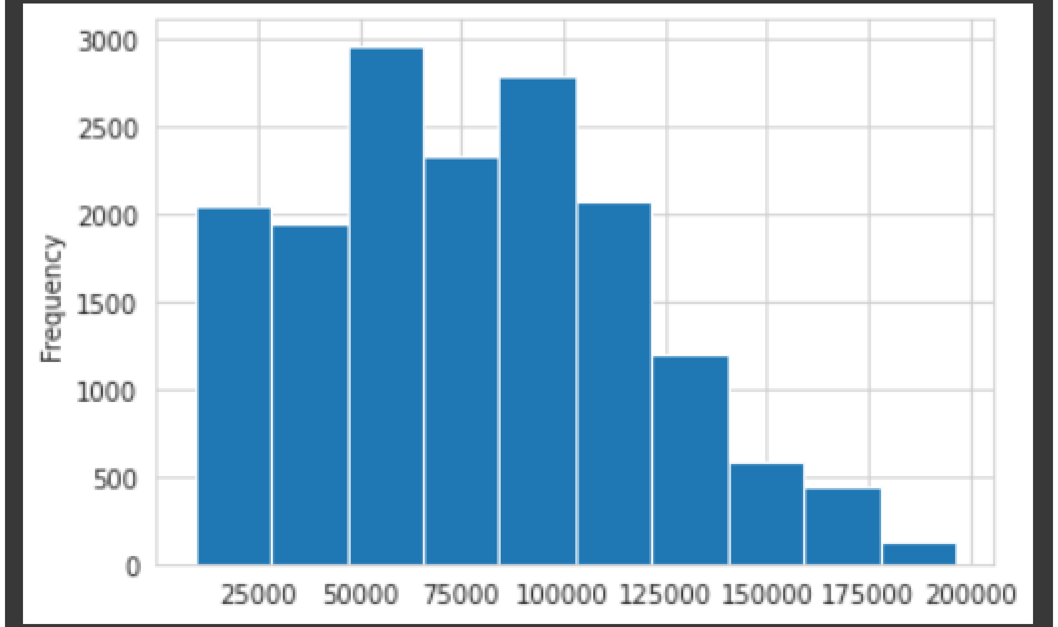
DATA EXPLORATION



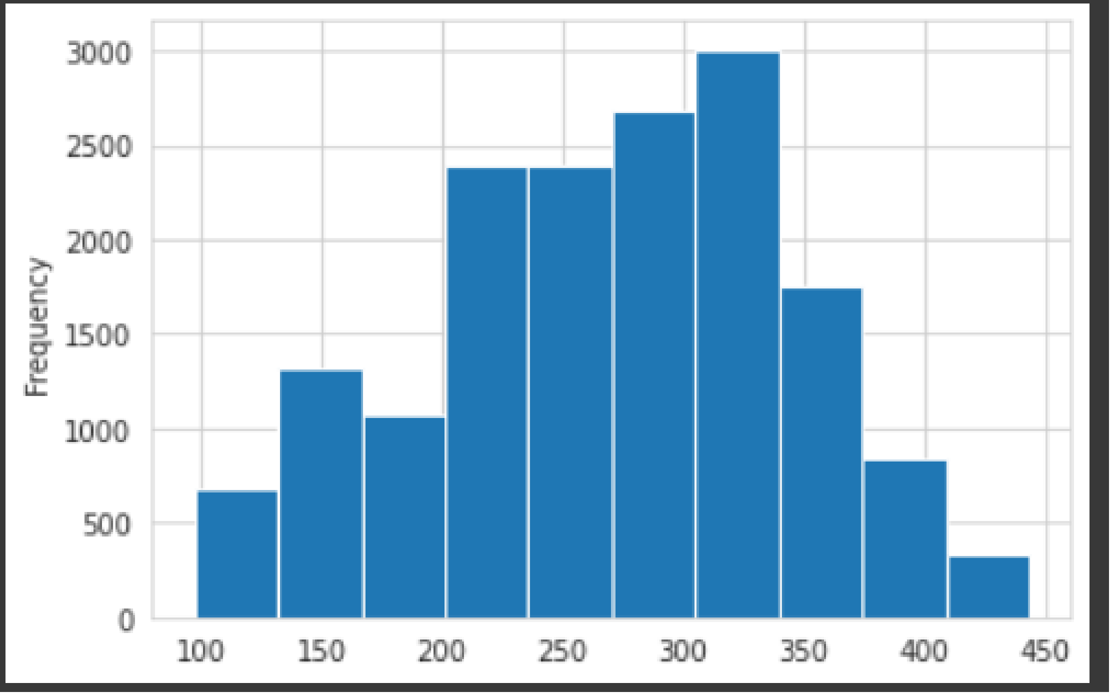


PRE-PROCESSING AND TRAINING DATA DEVELOPMENT

```
AWS['YearlyIncome'].plot.hist();
```




```
AWS['SqrtYearlyIncome'].plot.hist();
```



PREPROCESSING AND TRAINING DATA DEVELOPMENT

```
[ ]  # Let's one hot encode the categorical features

     def encode_string(cat_features):
         ## First encode the strings to numeric categories
         enc = preprocessing.LabelEncoder()
         enc.fit(cat_features)
         enc_cat_features = enc.transform(cat_features)
         ## Now, apply one hot encoding
         ohe = preprocessing.OneHotEncoder()
         encoded = ohe.fit(enc_cat_features.reshape(-1,1))
         return encoded.transform(enc_cat_features.reshape(-1,1)).toarray()

     categorical_columns = cat_features[1:]

     Features = encode_string(AWS[cat_features[0]])
     for col in categorical_columns:
         temp = encode_string(AWS[col])
         Features = np.concatenate([Features, temp], axis = 1)

     print(Features.shape)
     print(Features[:2, :])
```

```
(16471, 34)
[[1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1.
  0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

PREPROCESSING AND TRAINING DATA DEVELOPMENT

```
sel = fs.VarianceThreshold(threshold=(.8 * (1 - .8)))
Features_reduced = sel.fit_transform(Features)
```

```
[ ] ## Print the support and shape for the transformed features
print(sel.get_support())
print(Features_reduced.shape)
```

```
[ True False False  True False False False False  True  True  True  True
  True  True  True  True  True  True  True False  True False False False
 False False False  True  True  True False False False False False  True]
(16471, 18)
```

PREPROCESSING AND TRAINING DATA DEVELOPMENT

```
## Randomly sample cases to create independent training and validation data
nr.seed(9988)
indx = range(Features_reduced.shape[0])
indx = ms.train_test_split(indx, test_size = val_size)
X_train = Features[indx[0],:]
y_train = np.ravel(labels[indx[0]])
X_val = Features[indx[1],:]
y_val = np.ravel(labels[indx[1]])
```

```
[ ] scaler = preprocessing.StandardScaler().fit(X_train[:, 17:])
    X_train[:, 17:] = scaler.transform(X_train[:, 17:])
    X_val[:, 17:] = scaler.transform(X_val[:, 17:])
    X_train[:2,]
```

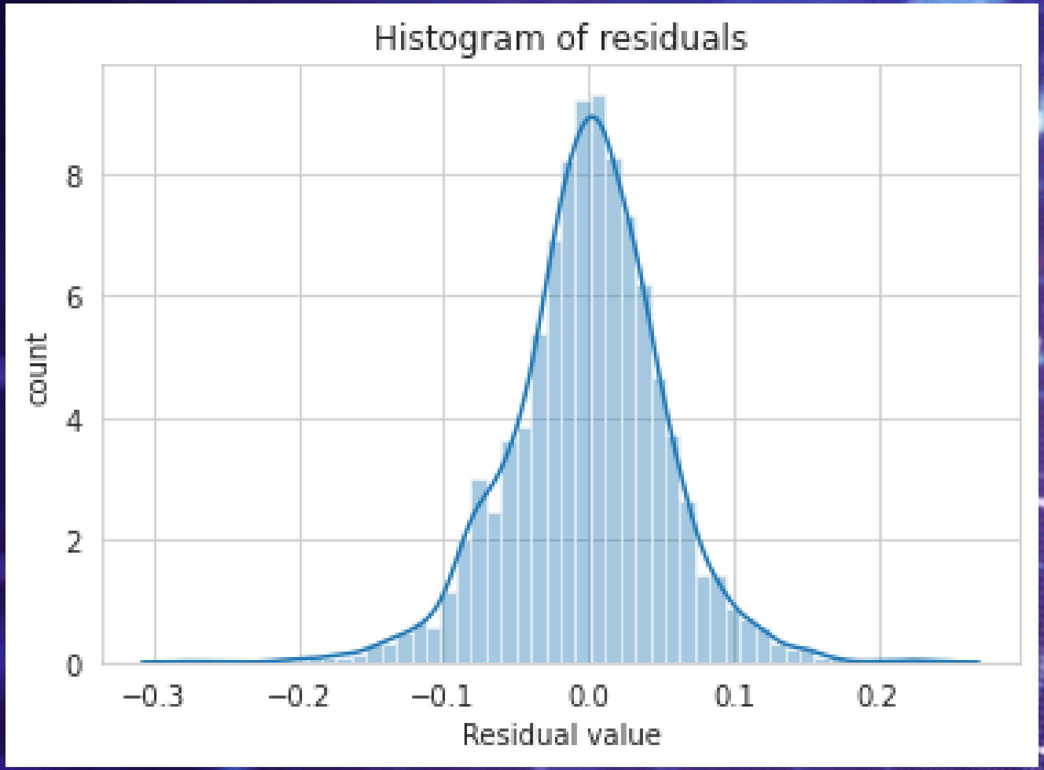
MODELING

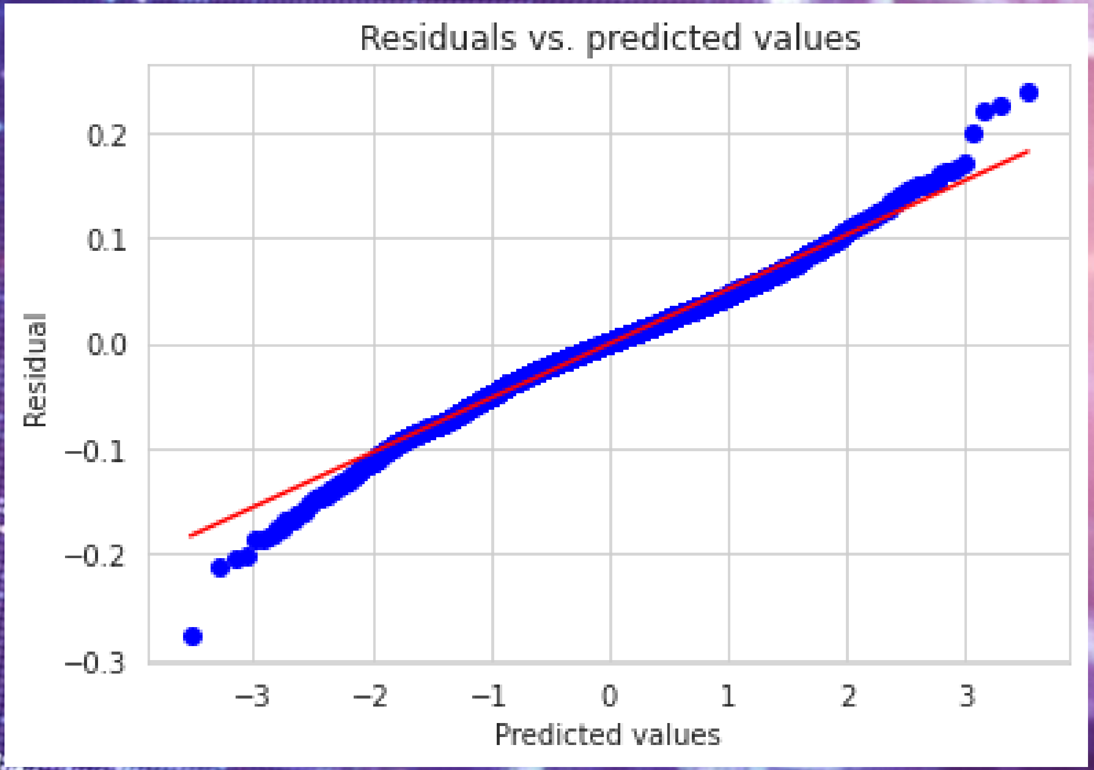
```
## define and fit the linear regression model
lin_mod = linear_model.LinearRegression()
lin_mod.fit(X_train, y_train)
```

DOCUMENTATION

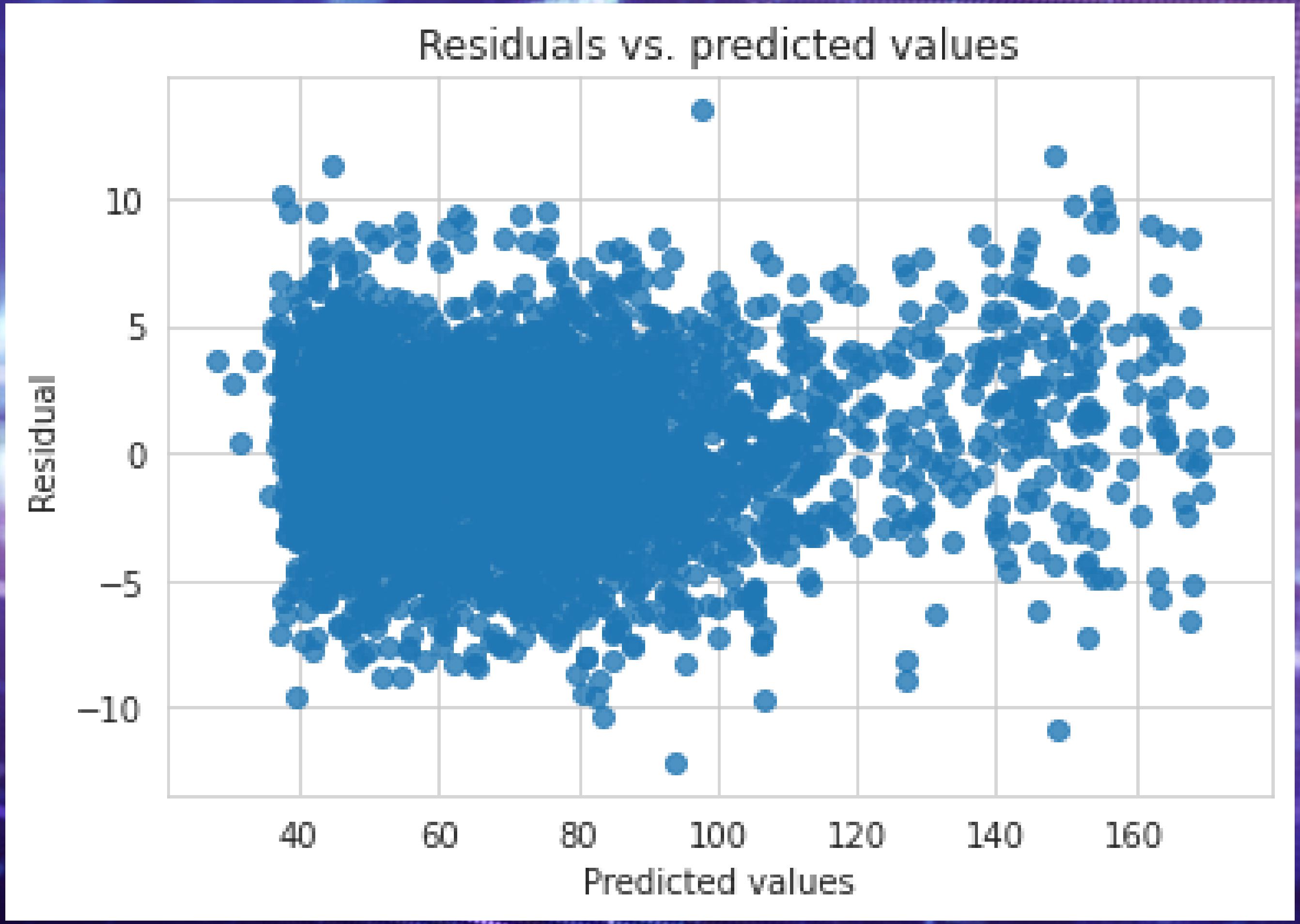
```
Mean Square Error          = 0.0027011334219300062
Root Mean Square Error     = 0.05197242944032929
Mean Absolute Error        = 0.03934928217789416
Median Absolute Error      = 0.0306460764547678
R^2                        = 0.9773796846795533
Adjusted R^2               = 0.977262302090894
```

DOCUMENTATION

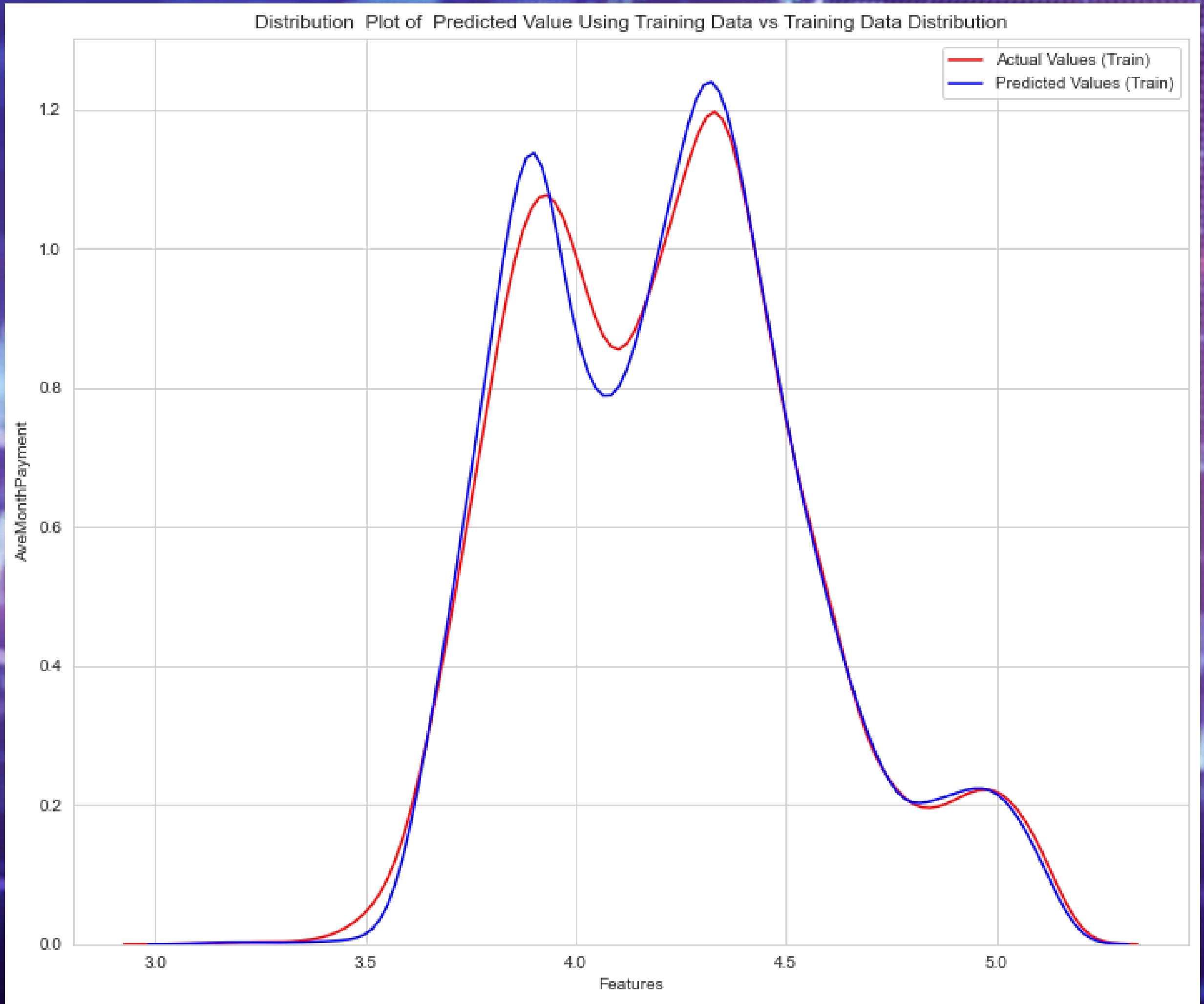




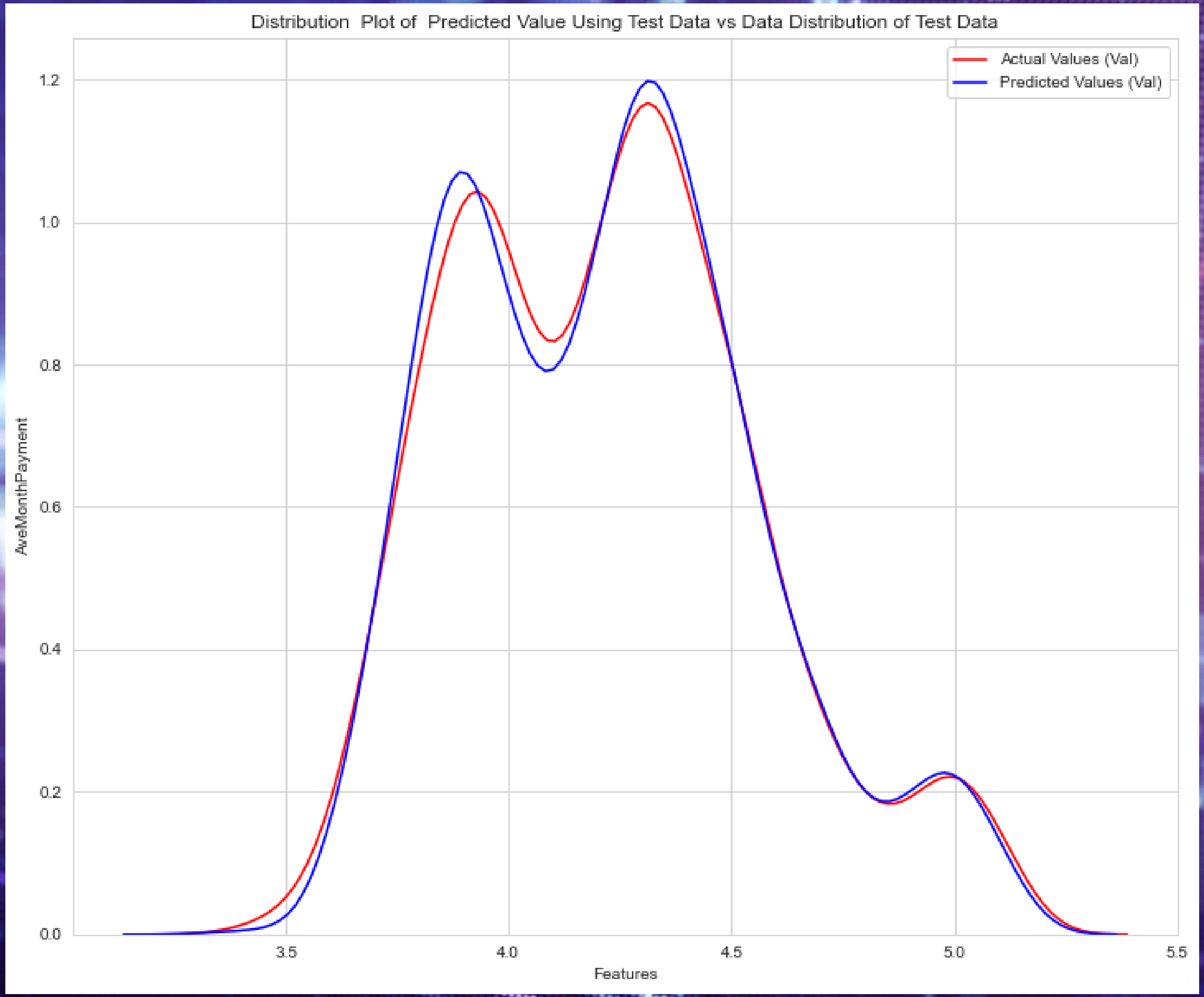
DOCUMENTATION



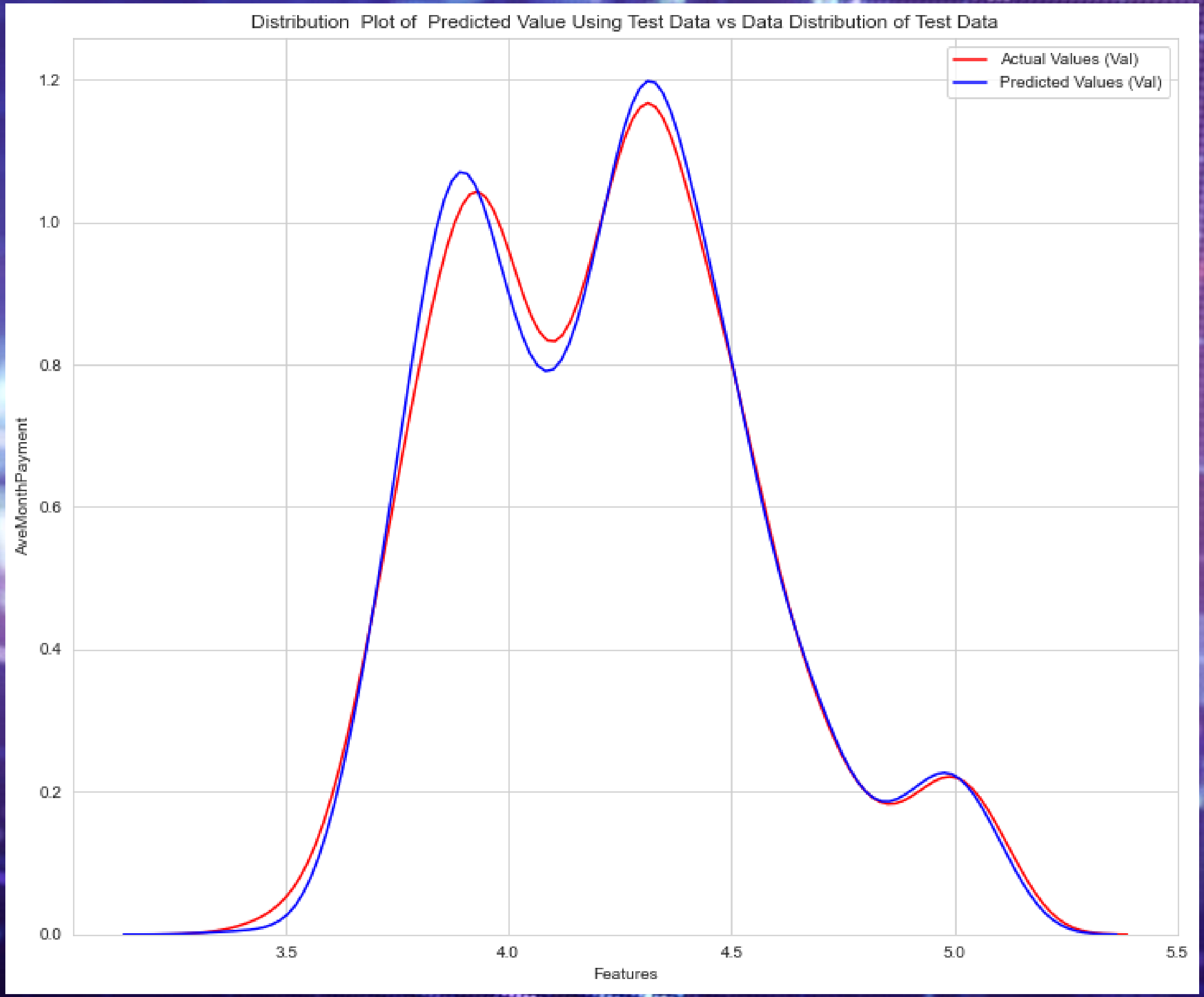
DOCUMENTATION



DOCUMENTATION



DOCUMENTATION



ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfpl = tfp.layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Ignore harmless warnings
import warnings
warnings.filterwarnings("ignore")

print('TF version:', tf.__version__)
print('TFP version:', tfp.__version__)
```

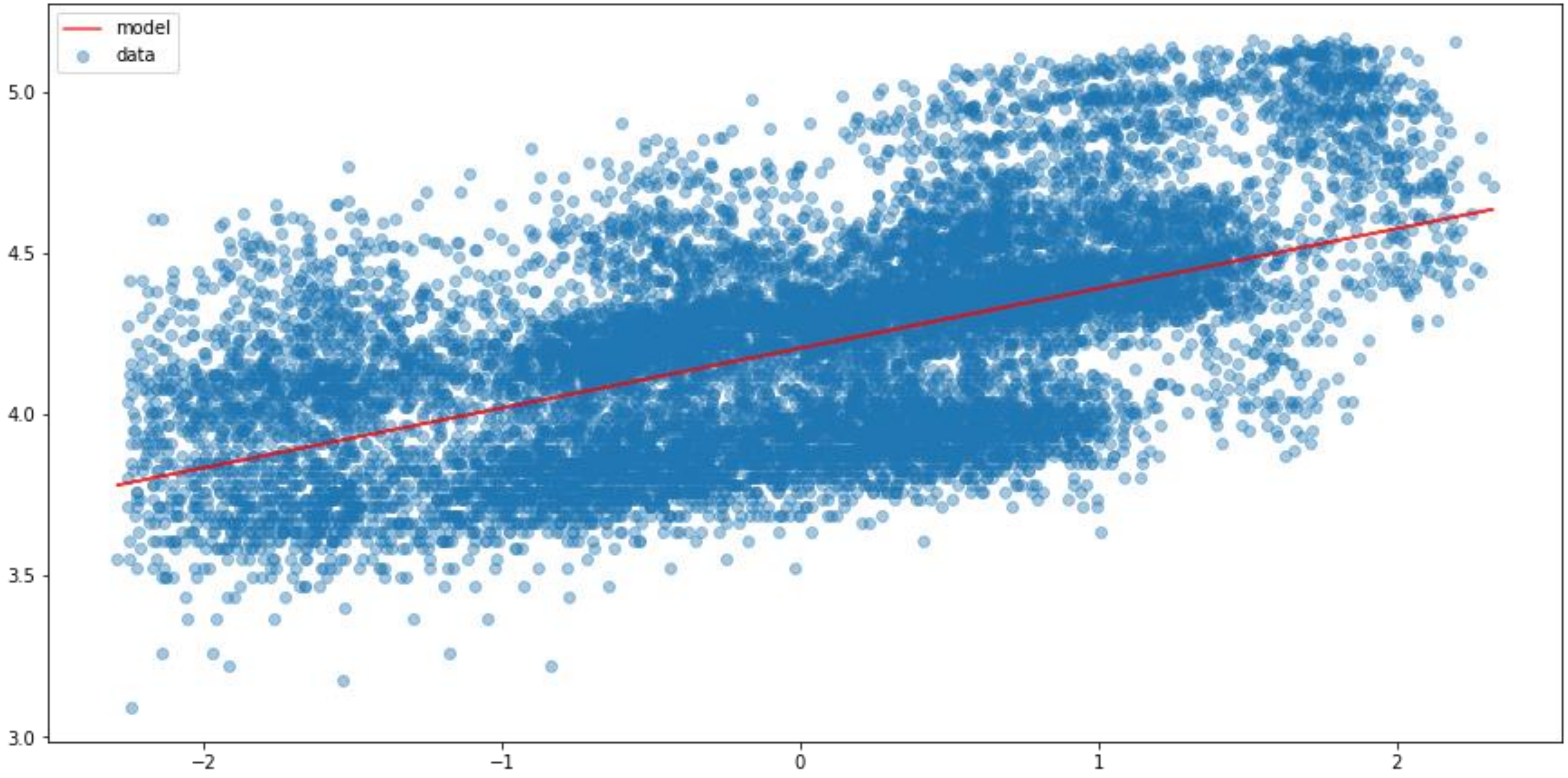
```
TF version: 2.4.1
TFP version: 0.12.1
```

ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

```
# Create and train deterministic linear model using mean squared error loss
# this is equivalent to our sci-kit learn model
# we will use one predictor variable in order to visualize in two dimensions

model = Sequential([
    Dense(units=1, input_shape=(1,))
])
model.compile(loss=MeanSquaredError(), optimizer=RMSprop(learning_rate=0.005))
model.summary()
model.fit(x_train_one, y_train, epochs=200, verbose=False)

# Plot the data and model
plt.figure(figsize=(15, 7.5))
plt.scatter(x_train_one, y_train, alpha=0.4, label='data')
plt.plot(x_train_one, model.predict(x_train_one), color='red', alpha=0.8, label='model')
plt.legend()
plt.show()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
```

ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)



ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

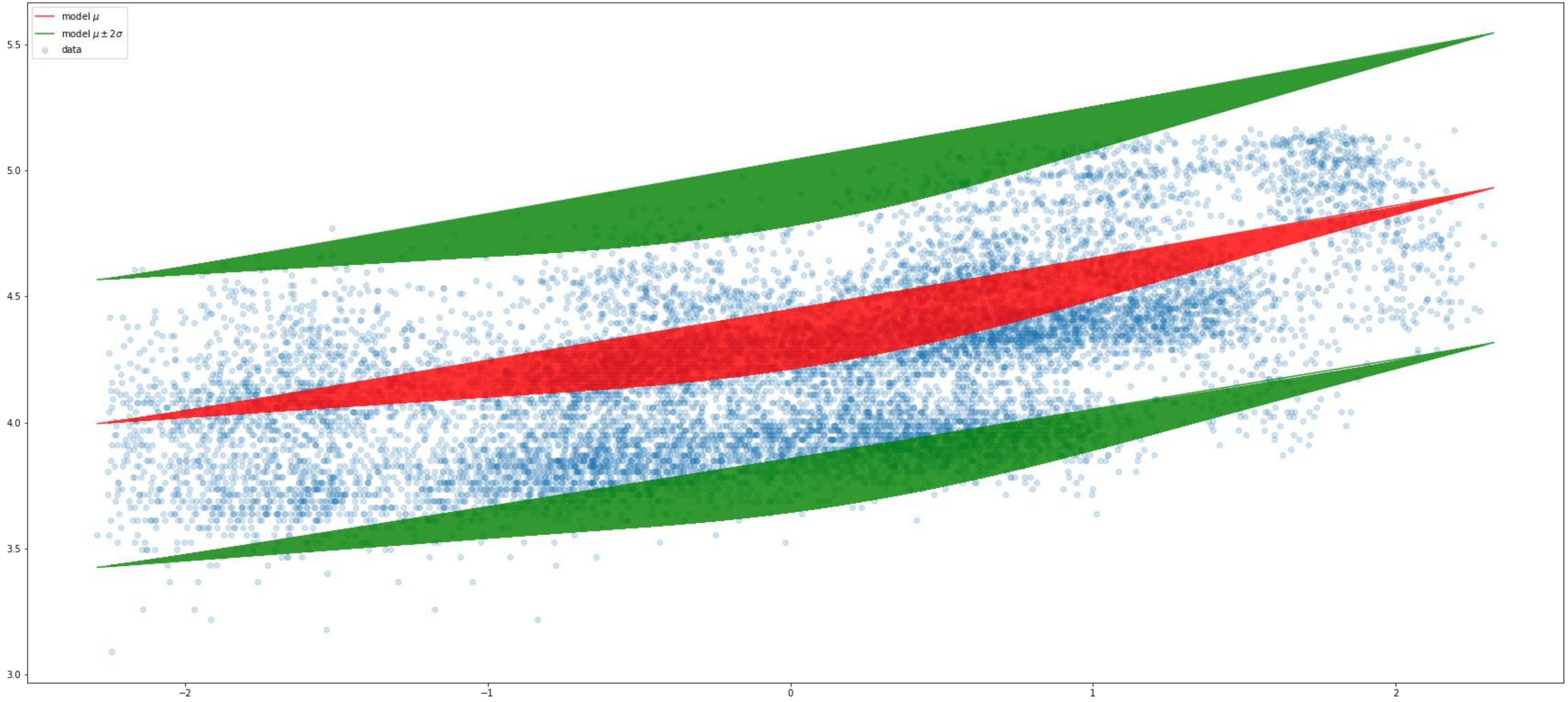
```
[ ] # Create negative log likelihood loss function

def nll(y_true, y_pred):
  """negative log likelihood"""
  return -y_pred.log_prob(y_true)
```

```
# Create probabilistic regression: normal distribution with fixed variance

model = Sequential([
    Dense(input_shape=(1,), units=8, activation='sigmoid'),
    Dense(tfpl.IndependentNormal.params_size(event_shape=1)),
    tfpl.IndependentNormal(event_shape=1)
])
model.compile(loss=nll, optimizer=RMSprop(learning_rate=0.01))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8)                 16
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 18
_________________________________________________________________
independent_normal (Independ multiple                  0
=================================================================
Total params: 34
Trainable params: 34
Non-trainable params: 0
_________________________________________________________________
```

ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

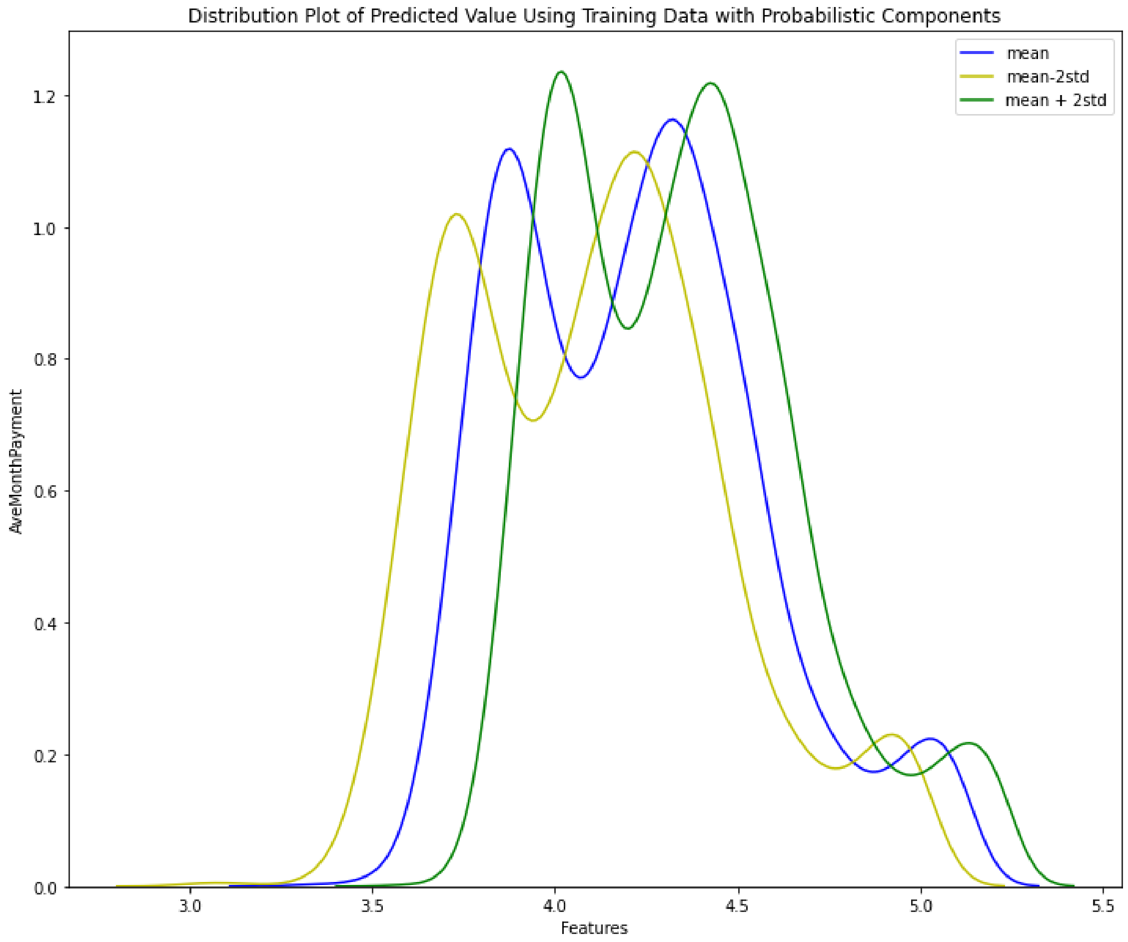


ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

```
# Create probabilistic regression: normal distribution with fixed variance
# All features

model2 = Sequential([
    Dense(input_shape=(36,), units=8, activation='sigmoid'),
    Dense(tfpl.IndependentNormal.params_size(event_shape=1)),
    tfpl.IndependentNormal(event_shape=1)
])
model2.compile(loss=nll, optimizer=RMSprop(learning_rate=0.01))
model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 8)                 296
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 18
_________________________________________________________________
independent_normal_1 (Indepe multiple                  0
=================================================================
Total params: 314
Trainable params: 314
Non-trainable params: 0
_________________________________________________________________
```

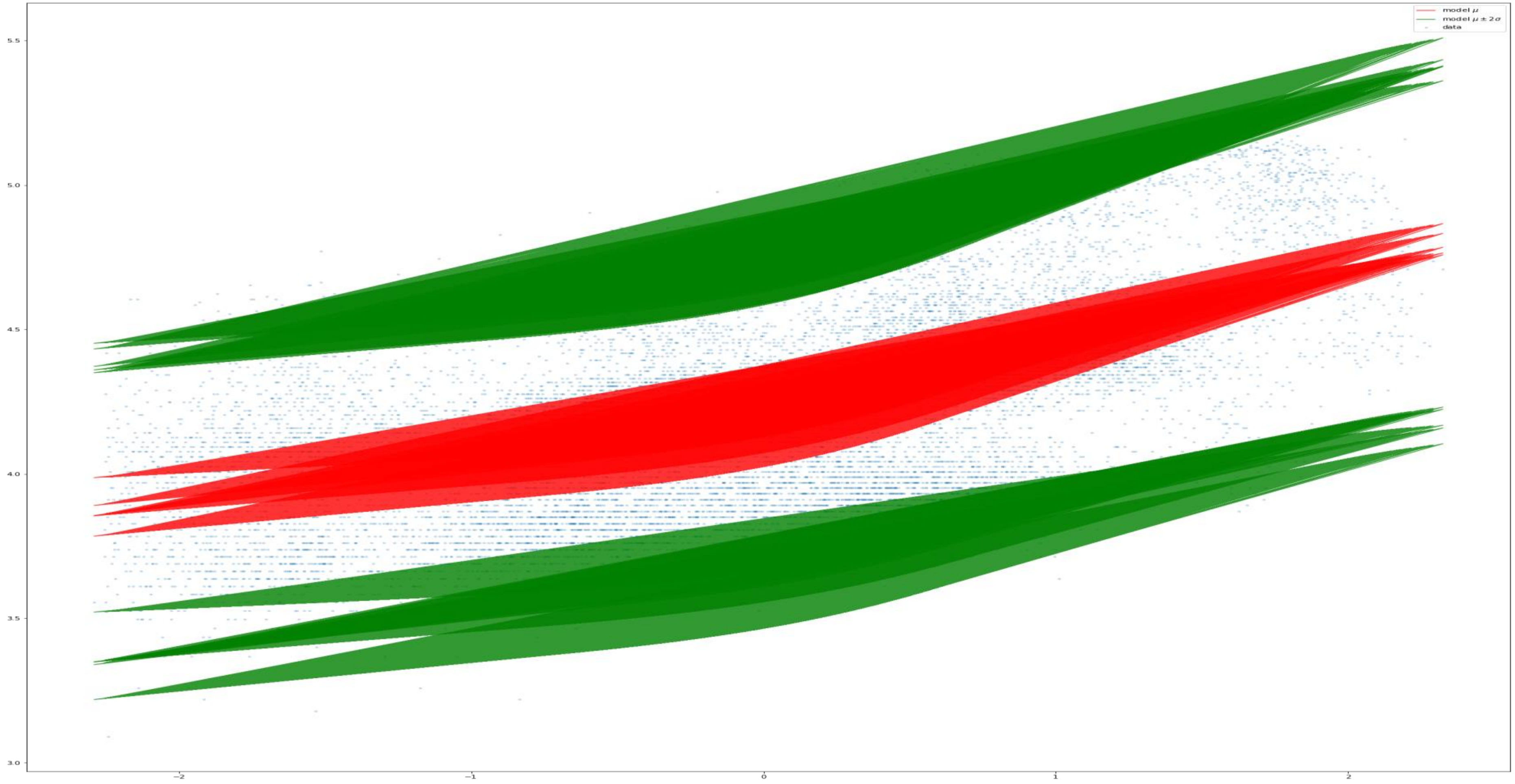
ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)



ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)

```
[ ]  # Create probabilistic regression with one hidden layer, weight uncertainty

     model3 = Sequential([
         tfpl.DenseVariational(units=8,
                               input_shape=(1,),
                               make_prior_fn=prior,
                               make_posterior_fn=posterior,
                               kl_weight=1/X_train.shape[0],
                               activation='sigmoid'),
         tfpl.DenseVariational(units=tfpl.IndependentNormal.params_size(1),
                               make_prior_fn=prior,
                               make_posterior_fn=posterior,
                               kl_weight=1/X_train.shape[0]),
         tfpl.IndependentNormal(1)
     ])



     model3.compile(loss=nll, optimizer=RMSprop(learning_rate=0.005))
     model3.summary()
```

ALTERNATIVE MODELING WITH UNCERTAINTY (RESEARCH)



SUMMARY

TensorFlow Probability provides a way to model while adding a measurement to the uncertainty of our models

Probabilistic modeling allows us to know where we are making assumptions and how wide of a range of values is contained in our 95% confidence interval for example

This project presents a set of ideas for future adoption of these methods in applied models

Visualization of the 95% confidence interval as a distribution using the Kernel Density Function for a fixed variance probabilistic model

Predictions can be served as a tuple, containing the 95% CI of each prediction

Further experiments can be performed for both evaluation of these methods and deployment scenarios

In the future we can perhaps study taking the average of the mean and standard deviation of the model with weight uncertainty (variational layers) over a significant number of model runs (we used 5 in the example but with computational power we can perhaps build more robust estimates.

THANK YOU!

Questions ?