1.1: Importing Libraries & Helper Functions

First of all, we will need to import some libraries and helper functions. This includes TensorFlow and some utility functions that I've written to save time.

In [35]:
```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

from utils import *
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, LambdaCallback

%matplotlib inline

print('Libraries imported.')
```

```
Libraries imported.
```

Task 2: Importing the Data

2.1: Importing the Data

The dataset is saved in a `data.csv` file. We will use `pandas` to take a look at some of the rows.

In [36]:
```
df = pd.read_csv('data.csv', names = column_names)
df.head()
```

Out[36]:

	serial	date	age	distance	stores	latitude	longitude	price
0	0	2009	21	9	6	84	121	14264
1	1	2007	4	2	3	86	121	12032
2	2	2016	18	3	7	90	120	13560
3	3	2002	13	2	2	80	128	12029
4	4	2014	25	5	8	81	122	14157

2.2: Check Missing Data

It's a good practice to check if the data has any missing values. In real world data, this is quite common and must be taken care of before any data pre-processing or model training.

In [37]:
```
df.isna().sum()
```

Out[37]:
```
serial      0
date        0
age         0
distance    0
```

```
stores       0
latitude     0
longitude    0
price        0
dtype: int64
```

Task 3: Data Normalization

3.1: Data Normalization

We can make it easier for optimization algorithms to find minimas by normalizing the data before training a model.

In [38]:

```
df = df.iloc[:,1:]
df_norm = (df - df.mean()) / df.std()
df_norm.head()
```

Out[38]:

	date	age	distance	stores	latitude	longitude	price
0	0.015978	0.181384	1.257002	0.345224	-0.307212	-1.260799	0.350088
1	-0.350485	-1.319118	-0.930610	-0.609312	0.325301	-1.260799	-1.836486
2	1.298598	-0.083410	-0.618094	0.663402	1.590328	-1.576456	-0.339584
3	-1.266643	-0.524735	-0.930610	-0.927491	-1.572238	0.948803	-1.839425
4	0.932135	0.534444	0.006938	0.981581	-1.255981	-0.945141	0.245266

3.2: Convert Label Value

Because we are using normalized values for the labels, we will get the predictions back from a trained model in the same distribution. So, we need to convert the predicted values back to the original distribution if we want predicted prices.

In [39]:

```
y_mean = df['price'].mean()
y_std = df['price'].std()

def convert_label_value(pred):
    return int(pred * y_std + y_mean)

print(convert_label_value(0.350088))
```

```
14263
```

Task 4: Create Training and Test Sets

4.1: Select Features

Make sure to remove the column **price** from the list of features as it is the label and should not be used as a feature.

In [40]:
```
X = df_norm.iloc[:, :6]
X.head()
```

Out[40]:

	date	age	distance	stores	latitude	longitude
0	0.015978	0.181384	1.257002	0.345224	-0.307212	-1.260799
1	-0.350485	-1.319118	-0.930610	-0.609312	0.325301	-1.260799
2	1.298598	-0.083410	-0.618094	0.663402	1.590328	-1.576456
3	-1.266643	-0.524735	-0.930610	-0.927491	-1.572238	0.948803
4	0.932135	0.534444	0.006938	0.981581	-1.255981	-0.945141

4.2: Select Labels

In [41]:
```
Y = df_norm.iloc[:, -1]
Y.head()
```

Out[41]:
```
0    0.350088
1   -1.836486
2   -0.339584
3   -1.839425
4    0.245266
Name: price, dtype: float64
```

4.3: Feature and Label Values

We will need to extract just the numeric values for the features and labels as the TensorFlow model will expect just numeric values as input.

In [42]:
```
X_arr = X.values
Y_arr = Y.values

print('X_arr shape: ', X_arr.shape)
print('Y_arr shape: ', Y_arr.shape)
```

```
X_arr shape:  (5000, 6)
Y_arr shape:  (5000,)
```

4.4: Train and Test Split

We will keep some part of the data aside as a **test** set. The model will not use this set during training and it will be used only for checking the performance of the model in trained and un-trained states. This way, we can make sure that we are going in the right direction with our model training.

In [43]:
```
X_train, X_test, y_train, y_test = train_test_split(X_arr, Y_arr, test_size = 0.05, shu

print('X_train shape: ', X_train.shape)
print('y_train shape: ', y_train.shape)
print('X_test shape: ', X_test.shape)
print('y_test shape: ', y_test.shape)
```

```
X_train shape:  (4750, 6)
y_train shape:  (4750,)
```

```
X_test shape:  (250, 6)
y_test shape:  (250,)
```

Task 5: Create the Model

5.1: Create the Model

Let's write a function that returns an untrained model of a certain architecture.

In [44]:
```
def get_model():

    model = Sequential([
        Dense(10, input_shape = (6,), activation = 'relu'),
        Dense(20, activation = 'relu'),
        Dense(5, activation = 'relu'),
        Dense(1)
    ])

    model.compile(
        loss='mse',
        optimizer='adadelta'
    )

    return model

model = get_model()
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 10)                70
_________________________________________________________________
dense_17 (Dense)             (None, 20)                220
_________________________________________________________________
dense_18 (Dense)             (None, 5)                 105
_________________________________________________________________
dense_19 (Dense)             (None, 1)                 6
=================================================================
Total params: 401
Trainable params: 401
Non-trainable params: 0
_________________________________________________________________
```

Task 6: Model Training

6.1: Model Training

We can use an `EarlyStopping` callback from Keras to stop the model training if the validation loss stops decreasing for a few epochs.

In [45]:
```
early_stopping = EarlyStopping(monitor='val_loss', patience = 5)

model = get_model()
```

```
preds_on_untrained = model.predict(X_test)

history = model.fit(
    X_train, y_train,
    validation_data = (X_test, y_test),
    epochs = 1000,
    callbacks = [early_stopping]
)
```

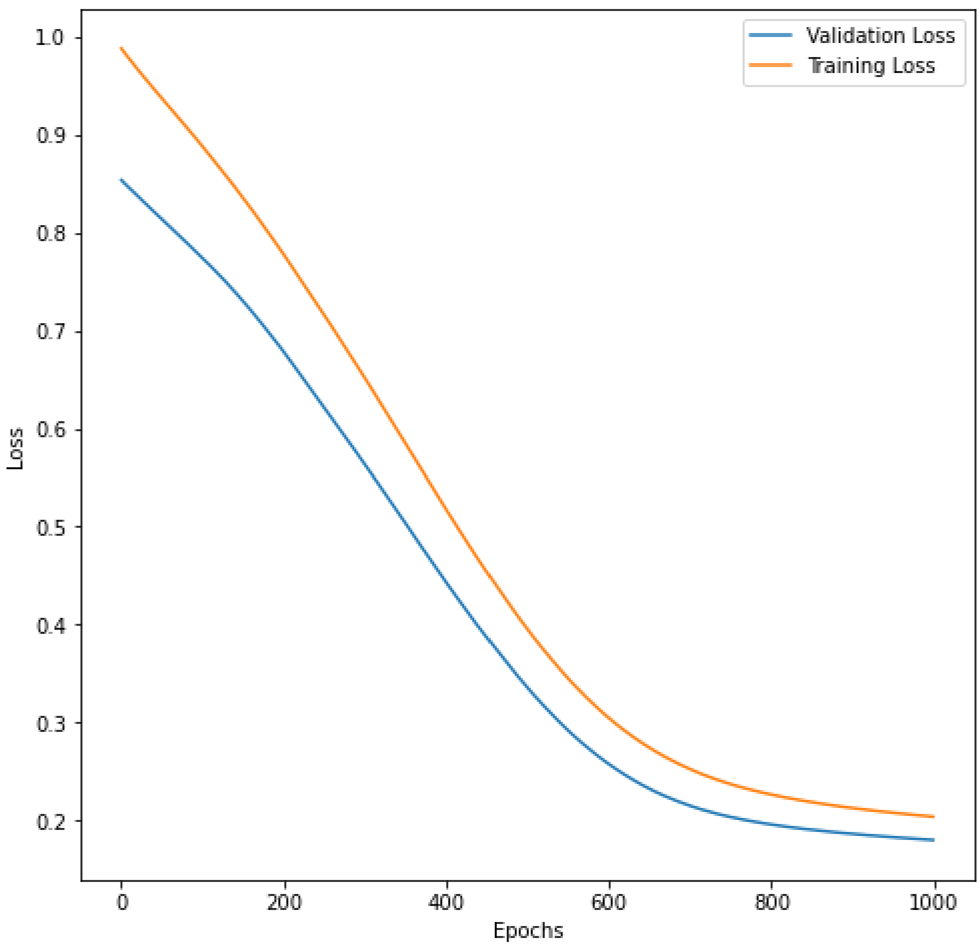
```
Train on 4750 samples, validate on 250 samples
Epoch 1/1000
4750/4750 [==============================] - 1s 149us/sample - loss: 0.9882 - val_loss:
0.8536
Epoch 2/1000
4750/4750 [==============================] - 0s 57us/sample - loss: 0.9870 - val_loss:
0.8527
Epoch 3/1000
4750/4750 [==============================] - 0s 58us/sample - loss: 0.9858 - val_loss:
0.8519
Epoch 4/1000
4750/4750 [==============================] - 0s 62us/sample - loss: 0.9847 - val_loss:
0.8510
Epoch 5/1000
4750/4750 [==============================] - 0s 67us/sample - loss: 0.9836 - val_loss:
0.8502
Epoch 6/1000
4750/4750 [==============================] - 0s 66us/sample - loss: 0.9825 - val_loss:
0.8494
Epoch 7/1000
4750/4750 [==============================] - 0s 55us/sample - loss: 0.9815 - val_loss:
0.8486
Epoch 8/1000
4750/4750 [==============================] - 0s 66us/sample - loss: 0.9804 - val_loss:
0.8478
Epoch 9/1000
4750/4750 [==============================] - 0s 49us/sample - loss: 0.9793 - val_loss:
0.8469
Epoch 10/1000
4750/4750 [==============================] - 0s 54us/sample - loss: 0.9782 - val_loss:
0.8461
Epoch 11/1000
4750/4750 [==============================] - 0s 67us/sample - loss: 0.9772 - val_loss:
0.8453
Epoch 12/1000
4750/4750 [==============================] - 0s 61us/sample - loss: 0.9761 - val_loss:
0.8445
Epoch 13/1000
4750/4750 [==============================] - 0s 59us/sample - loss: 0.9751 - val_loss:
0.8437
Epoch 14/1000
4750/4750 [==============================] - 0s 56us/sample - loss: 0.9740 - val_loss:
0.8429
Epoch 15/1000
4750/4750 [==============================] - 0s 65us/sample - loss: 0.9730 - val_loss:
0.8421
Epoch 16/1000
4750/4750 [==============================] - 0s 54us/sample - loss: 0.9719 - val_loss:
0.8412
Epoch 17/1000
4750/4750 [==============================] - 0s 54us/sample - loss: 0.9709 - val_loss:
0.8404
Epoch 18/1000
4750/4750 [==============================] - 0s 54us/sample - loss: 0.9698 - val_loss:
0.8396
Epoch 19/1000
```

```
4750/4750 [==============================] - 0s 57us/sample - loss: 0.2043 - val_loss:
 0.1803
Epoch 995/1000
4750/4750 [==============================] - 0s 64us/sample - loss: 0.2042 - val_loss:
 0.1803
Epoch 996/1000
4750/4750 [==============================] - 0s 66us/sample - loss: 0.2042 - val_loss:
 0.1802
Epoch 997/1000
4750/4750 [==============================] - 0s 62us/sample - loss: 0.2041 - val_loss:
 0.1802
Epoch 998/1000
4750/4750 [==============================] - 0s 66us/sample - loss: 0.2040 - val_loss:
 0.1801
Epoch 999/1000
4750/4750 [==============================] - 0s 56us/sample - loss: 0.2039 - val_loss:
 0.1801
Epoch 1000/1000
4750/4750 [==============================] - 0s 62us/sample - loss: 0.2039 - val_loss:
 0.1800
```

6.2: Plot Training and Validation Loss

Let's use the `plot_loss` helper function to take a look training and validation loss.

In [46]:
```
plot_loss(history)
```



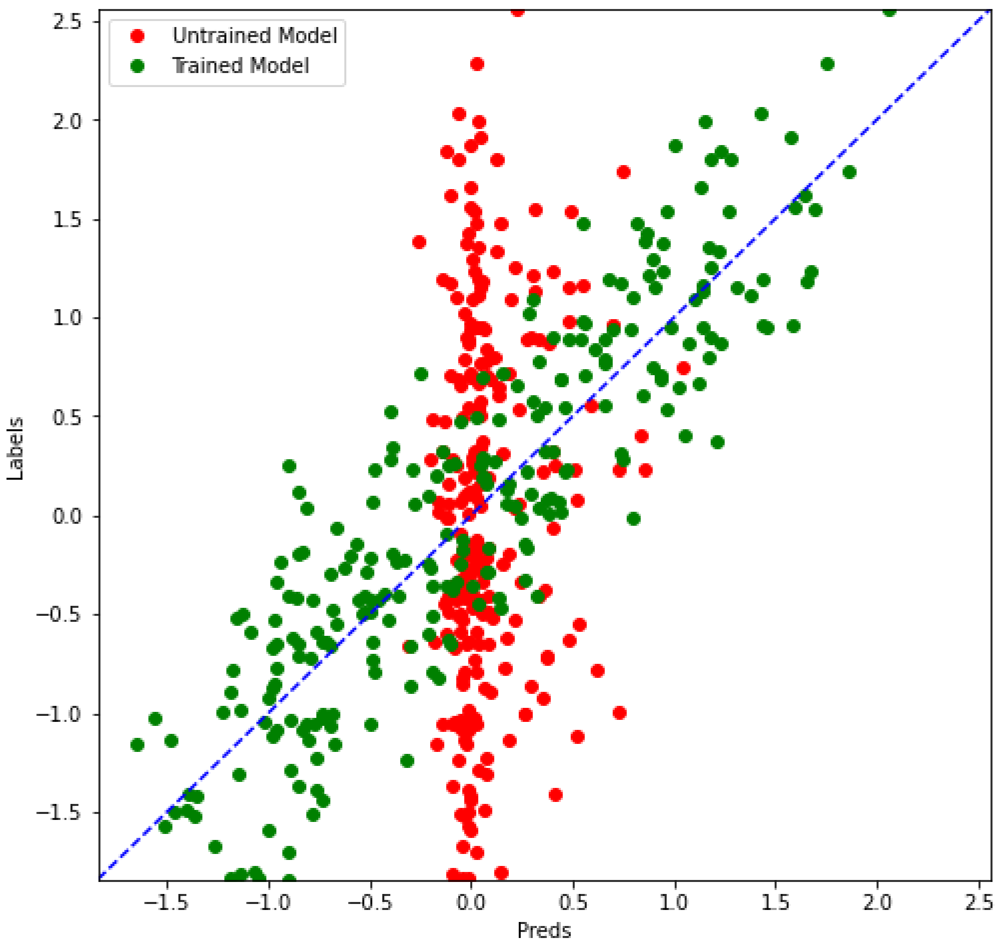
Task 7: Predictions

7.1: Plot Raw Predictions

Let's use the `compare_predictions` helper function to compare predictions from the model when it was untrained and when it was trained.

In [47]:
```
preds_on_trained = model.predict(X_test)

compare_predictions(preds_on_untrained, preds_on_trained, y_test)
```

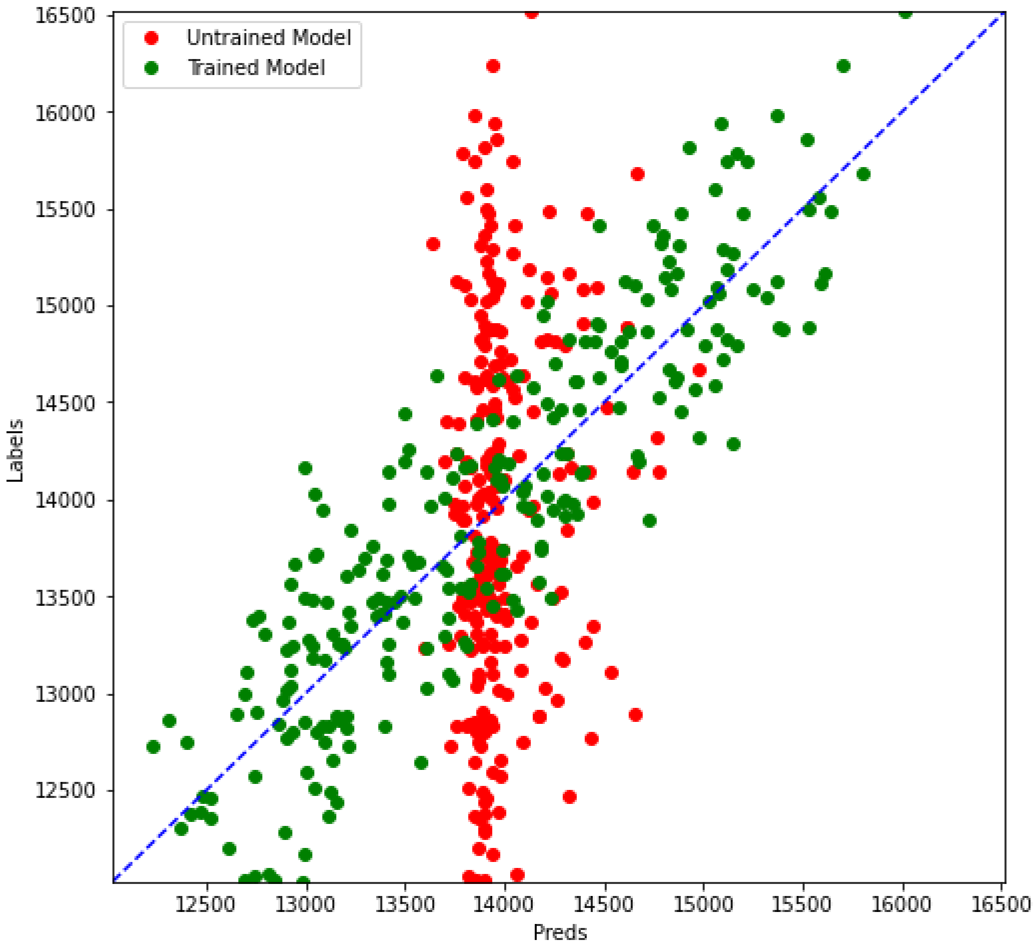


7.2: Plot Price Predictions

The plot for price predictions and raw predictions will look the same with just one difference: The x and y axis scale is changed.

In [48]:
```
price_on_untrained = [convert_label_value(y) for y in preds_on_untrained]
price_on_trained = [convert_label_value(y) for y in preds_on_trained]
price_y_test = [convert_label_value(y) for y in y_test]

compare_predictions(price_on_untrained, price_on_trained, price_y_test)
```


Untrained Model
Trained Model
Labels
Preds
16500
16000
15500
15000
14500
14000
13500
13000
12500
12500
13000
13500
14000
14500
15000
15500
16000
16500

In [ ]: