RSRP Regression Model Report

Muhammad Hassaan Chaudhary

June 2025

Contents

1 Datasets Used

1.1 California Housing Dataset

The California Housing dataset is a well-known open-source dataset provided by `scikit-learn`. It contains housing data from California in the 1990s, including information such as median income, average house age, and average number of rooms, along with the target variable — the median house value for California districts. I utilized this dataset to practice regression model building in a reproducible and verifiable environment.

1.2 Signal Strength Dataset

The Signal Strength dataset used in this report is a private dataset provided for the purpose of this project. It includes raw network data from a simulated mobile communication environment, including RSRP (Reference Signal Received Power) readings and various network-related features. Due to the private nature of the data, it cannot be publicly shared, but all preprocessing, feature extraction, and modeling steps are described in full.

2 Prelude: Building a basic Linear Regression model using Keras

My first task was to train a basic regression model on the signal strength dataset, plot feature importance, and compute RMSE. However, prior to working with the signal strength dataset, I chose to hone my skills by building a basic Linear Regression model using an open-source dataset. The benefits of such were that if I ran into any problems while building my model, I could refer to multiple resources that have previously worked with this open source dataset to find the solution to my problem.

The steps I followed to build this model were as follows:

1. Load the Dataset
2. Assign input (X) and output (y) variables
3. Split data set into training and testing sets
4. Scale model
5. Define RMSE function
6. Define model
7. Compile model
8. Train Model
9. Evaluate model on test set
10. Create predictions
11. Create visualization of Predicted values vs. Actual values

For this model, I used sklearn to source a dataset and complete preprocessing and Keras to build and train the model.

1. Load the Dataset: To load the dataset, I used sklearn's in house datasets. The dataset I used was comprised of California Housing Data.

```
        from sklearn.datasets import fetch_california_housing
```

2. Assign input (X) and output (y) variables: I assigned the dataset to the variable california and set the input (X) as california.data and the output (y) as california.target.

```
        X = california.data
        y = california.target
```

3. Then I split my dataset into training and test set:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
↪  test_size =
0.2, random_state=42)
```

4. Scale model: I scaled my input so the model trains more efficiently and fairly. I utilized the standard scaler because I am assuming the data in my dataset is roughly normally distributed.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

   (a) Scaling is an important step in building any linear model. Scaling is essentially transforming your data so that it fits within a certain range or distribution. It is done to make input variables more comparable in scale. There are different types of scaling: Standard Scaling, Min-Max Scaling, Robust Scaling. Standard Scaling is done when data is normally distributed or if you're using a model that assumes normality, like regression models or neural networks. Min-Max Scaling is done when data is bounded and you need features in a specific range. Finally, Robust Scaling is done when your data has many outliers and you want to reduce the influence of extreme values.

5. Define RMSE function: I defined RMSE as a function because it is not a built in function for Keras or sklearn.

```
import tensorflow.keras.backend as K
def rmse(y_true, y_pred):C
    return K.sqrt(K.mean(K.square(y_pred-y_true)))
```

6. Define model: I defined the model to have one neuron and linear activation. The one neuron part ensures the model outputs a single numeric value - the prediction. The linear activation part implies that the model has no hidden layers and no activation functions other than linear. Ultimately, these factors will ensure that this model will be functionally identical to a Linear Regression model

```
model = Sequential()
model.add(Dense(1, input_shape = (X_train.shape[1],),
↪  activation =
'linear'))
```

7. Compile model: I compiled the model to regulate how it will learn. Within this, I included my previously made RMSE function as a metric to track model performance

```
model.compile(optimizer='adam', loss = 'mse', metrics =
↪  [rmse,
'mae'])
```

8. Train Model: I trained the model using the model.fit function. Through this line of code, the model trained 100 times over the dataset, updated weights every 32 examples, tracked progress on a validation set, and returned a history object with metrics over time.

```
        history = model.fit(X_train, y_train, epochs=100,
        ↪  batch_size = 32,
        validation_split = 0.2, verbose = 1)
```

9. Then I evaluated how well the model performed on the test set.

```
        loss, test_rmse, test_mae = model.evaluate(X_test, y_test,
        ↪  verbose=0)
        print(f"Test RMSE:{test_rmse:.2f}")
        print(f"Test MAE:{test_mae:.2f}")
```

10. Then I created predictions of the target values.

```
        predictions = model.predict(X_test[:5]).flatten()
        print("Predictions:", predictions)
        print("Actual values", y_test[:5])
```

11. Finally, I created visualizations that depicted the predictions my model output vs. the actual values from the dataset.

```
        y_pred = model.predict(X_test).flatten()
        import matplotlib.pyplot as plt
        plt.figure(figsize = (8,6))
        plt.scatter(y_test, y_pred, alpha = 0.5, color = 'blue',
        ↪  label =
        'Predicted vs Actual')
        plt.plot([y_test.min(), y_test.max()], [y_test.min(),
        ↪  y_test.max()],
        'r--', label = 'Ideal Prediction')
        plt.xlabel("Actual Prices")
        plt.ylabel("Predicted Prices")
        plt.title("California Housing: Predictions vs Actuals")
        plt.legend()
        plt.grid(True)
        plt.show()
```

Resulting Data:

Metric	**Value**
Root Mean Squared Error (RMSE)	1.43
Mean Absolute Error (MAE)	0.53

Table 1: RMSE and MAE of Housing Dataset regression model

Sample	**Actual Value**	**Predicted Value**
1	0.477	0.709
2	0.458	1.755
3	5.000	2.704
4	2.186	2.838
5	2.780	2.592

Table 2: Housing Dataset: Predicted vs. Actual values

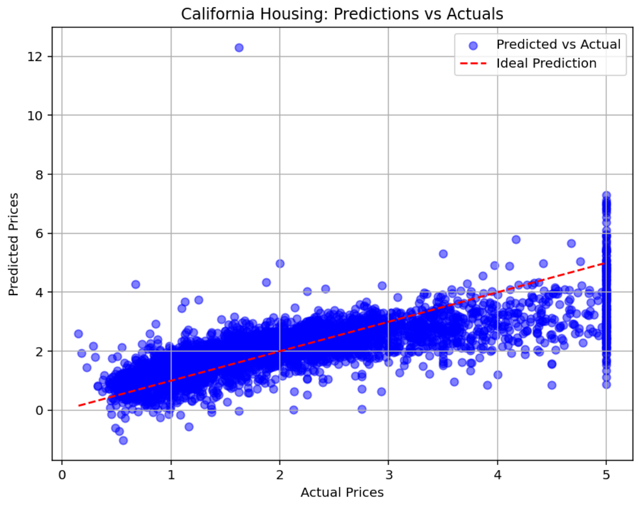

Figure 1: CA Housing Dataset: Predictions vs. Actuals

3 Performing feature extraction on Signal Strength Dataset

The previously shown Linear Regression model gave me a good understanding of building ML models going into working with the given Signal Strength dataset. Now I shifted my focus onto the given Signal Strength dataset. However, before I could input my dataset into a ML Model, I had to perform feature extraction onto the raw data to get it into a usable form. The steps I followed to complete feature extraction were as follows:

1. Load the Data: The first step was to load the raw data into the program. I chose to use first simulation as my raw data

```
        import pandas as pd
        df =
        ↪  pd.read_csv("/Users/muhammadhassaanchaudhary/Downloads/1_Simulation
        0_mobile.txt", encoding="latin-1", sep=',')
```

2. Initial cleaning of the data: I removed commas from columns which have them

```
        df = df.replace(',', '', regex=True)
```

3. Converting columns to numeric: This ensures that the data is all in a form that can be processed later by my model

```
        for col in df.columns:
                df[col] = pd.to_numeric(df[col], errors = 'coerce')
```

4. Identify column names: I identified all column names so I know the exact name of the target column to use when specifying the target column in the next step.

```
        print(df.columns.tolist())
```

5. Identify target column and keep it separate

```
    target_col = 'RSRP (DL)'
    if target_col not in df.columns:
            raise ValueError(f"Target Column '{target_col}' not
            ↪  found in dataset")
```

6. Drop rows where target is missing

```
        df = df[df[target_col].notna()]
```

7. Separate features (X) and target (y)

```
        X = df.drop(columns=[target_col])
        y = df[target_col]
```

8. Identify categorical columns: This was so that I know which columns to convert into numeric form so that my model can understand them.

```
    categorical_cols =
    ↪  X.select_dtypes(include='object').columns.tolist()
```

9. Encode categorical columns: Ensure all categorical columns are turned into numerical form

```
    X = pd.get_dummies(X, columns=categorical_cols,
    ↪  drop_first=True)
```

10. Handle missing values: Handle missing values by filling them with column mean so model does not encounter missing values.

```
        X = X.fillna(X.mean())
```

11. Combine cleaned features

```
        cleaned_df = pd.concat([X,y], axis=1)
```

12. Save to csv file

```

    ↪  cleaned_df.to_csv("/Users/muhammadhassaanchaudhary/Downloads/cleaned_RSRP
    _data.csv", index = False)
```

13. Print completion statement

```
        print("Data cleaned and saved with 'RSRP (DL)' as target")
```

4 Building a Linear Regression model using the Signal Strength Dataset

To build the Linear Regression model with the Signal Strength dataset I followed the same steps as I did earlier with the open source dataset, with the addition of a step to further clean the data. The steps were as follows:

1. Load Dataset

```
import pandas as pd
df =
↪  pd.read_csv("/Users/muhammadhassaanchaudhary/Downloads/cleaned_RSRP_data.csv")
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Standardscaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

2. Assign input (X) and output (y) variables: The intended dependent (y) variable was RSRP (DL).

```
y = df['RSRP (DL)']
X = df.drop('RSRP (DL)', axis = 1)
```

3. Last minute cleaning data: This was to ensure there was no nan values in X train. Prior to adding this line there were nan values in X train, which caused the model's predictions be nan.

```
X = X.replace([np.inf, -np.inf], np.nan)
X = X.dropna(axis=1, how='all')
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
X = pd.DataFrame(imputer.fit_transform(X), columns =
↪  X.columns)
```

4. Create training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=100)
```

5. Scale model: I scaledd my input so the model trains more efficiently and fairly.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

6. Double check for nan values in each training and test set

```
        print("Any NaNs in X_train?", np.isnan(X_train).any())
        print("Any Infs in X_train?", np.isinf(X_train).any())
        print("Any NaNs in y_train?", np.isnan(y_train).any())
        print("Any Infs in y_train?", np.isinf(y_train).any())
```

7. Define RMSE function: I defined RMSE as a function because it is not a built in funciton for Keras or sklearn.

```
        import tensorflow.keras.backend as K
        def rmse (y_true, y_pred):
            return K.sqrt(K.mean(K.square(y_pred-y_true)))
```

8. Define model: I defined the model to have one neuron and linear activation. The one neuron part ensures the model outputs a single numeric value - the prediction. The linear activation part implies that the model has no hidden layers and no activation functions other than linear. Ultimately, these factors will ensure that this model will be functionally identical to a Linear Regression model.

```
        from tensorflow.keras import Input
        model = Sequential()
        model.add(Input(shape=(X_train.shape[1],)))
        model.add(Dense(1, activation = 'linear'))
```

9. Compile model: I compiled the model to regulate how it will learn. Within this code, I included my RMSE function as a metric to track model performance

```
         model.compile(optimizer = 'adam', loss = 'mse', metrics =
          ↪  [rmse, 'mae'])
```

10. Train model: I ensured this model trained 100 times over the dataset, updated weights every 32 examples, tracked progress on a validation set, and returned a history object with metrics over time.

```
        history = model.fit(X_train, y_train, epochs = 100,
        ↪  batch_size = 32,
        validation_split = 0.2, verbose = 1)
```

11. Evaluate model on the test set

```
        loss, test_rmse, test_mae = model.evaluate(X_test, y_test,
        ↪  verbose
        = 0)
        print(f"Test RMSE:{test_rmse:.2f}")
        print(f"Test MAE:{test_mae:.2f}")
```

12. Create predictions of target values and print actual values for comparison.

```
        predictions = model.predict(X_test[:5]).flatten()
        print("Predictions:", predictions)
        print("Actual values", y_test[:5])
```

13. Create visualizations: I made a visualization to compare the predicted RSRP from my model and the actual RSRP values from the dataset. I also included a line of best fit that depicted the ideal prediction.

```
        y_pred = model.predict(X_test).flatten()
        import matplotlib.pyplot as plt
        plt.figure(figsize = (8,6))
        plt.scatter(y_test, y_pred, alpha = 0.5, color = 'blue',
        ↪  label =
        'Predicted vs Actual')
        plt.plot([y_test.min(), y_test.max()], [y_test.min(),
        ↪  y_test.max()],
        'r--', label = 'Ideal Prediction')
        plt.xlabel("Actual RSRP")
        plt.ylabel("Predicted RSRP")
        plt.title("RSRP in Cellular Networks: Predictions vs
        ↪  Actuals")
        plt.legend()
        plt.grid(True)
        plt.show()
```

Resulting Data:

Metric	Value
Root Mean Squared Error (RMSE)	73.74
Mean Absolute Error (MAE)	72.78

Table 3: RMSE and MAE of regression model

Actual Value	Prediction
-70.40	1.56
-80.46	-5.73
-70.90	3.49
-72.09	-1.19
-80.46	-7.36

Table 4: RSRP in Cellular Networks: Predictions vs Actuals

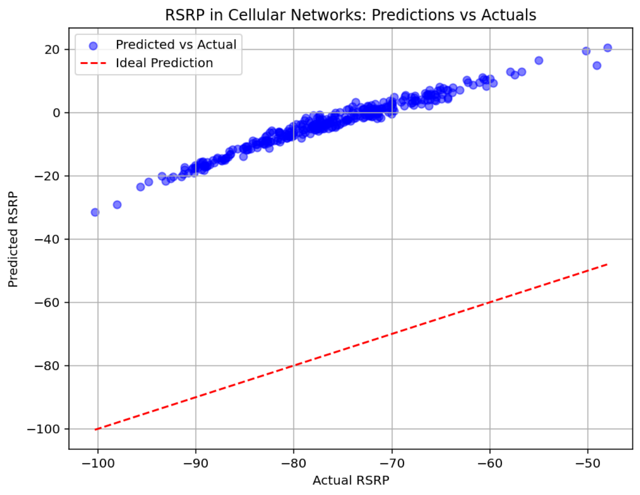

Figure 2: RSRP in Cellular Networks: Predictions vs. Actuals

5 Revising my model

Looking at my results, there was a large disparity between my predictions and the actual RSRP figures from the signal strength dataset.

To fix this, I chose to add polynomial features to my model. This addition is helpful as it helps the model account for curved relationships. If the cause for my high RMSE is that the relationship is not completely linear, this should account for that and lower the RMSE dramatically.

The code I used to accomplish this was as follows:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, interaction_only=False,
↪  include_bias=False)
X_poly = poly.fit_transform(X)
X = pd.DataFrame(X_poly,
↪  columns=poly.get_feature_names_out(X.columns))
```

Unfortunately, after doing so my predictions stayed the same and the RMSE did not lower.

Results post adding polynomial features:

Metric	**Value**
Root Mean Squared Error (RMSE)	72.77
Mean Absolute Error (MAE)	72.70

Table 5: RMSE and MAE of regression model post polynomial features addition

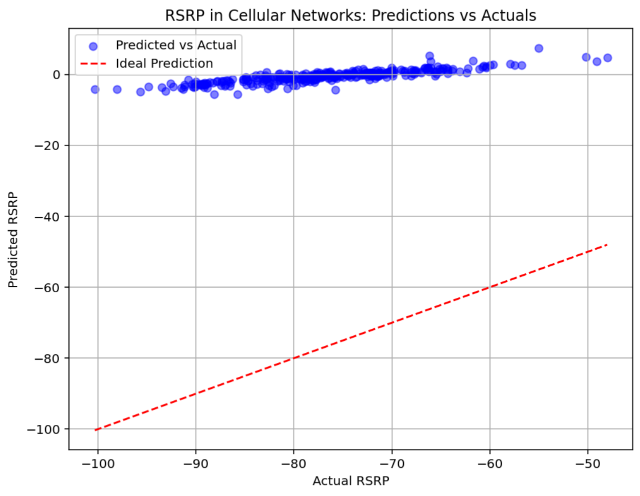

Figure 3: RSRP in Cellular Networks: Predictions vs Actuals (Post Poly-features

To fix this problem I carried out the following fixes to my code:

1. Removing low importance features: The polynomial features aspect I added earlier may have created to many useless combinations, so I removed low importance features to keep only the most predictive ones.

```
from sklearn.feature_selection import SelectKBest,
↪  f_regression
selector = SelectKBest(f_regression, k=20)
X_selected = selector.fit_transform(X, y)
selected_columns = X.columns[selector.get_support()]
X = pd.DataFrame(X_selected, columns=selected_columns)
```

2. Checking for feature colinearity and remove highly correlated features: I checked for feature colinearity in case multiple features were too highly correlated in which case I removed them. If multiple features are too highly correlated it is harder for the model to determine each features individual contribution.

```
import seaborn as sns
import matplotlib.pyplot as plt

corr_matrix = pd.DataFrame(X).corr().abs() #calculate
↪  correlation matrix
plt.figure(figsize=(12,8))
sns.heatmap(corr_matrix, cmap = 'coolwarm')
plt.title("Feature Correlation Matrix")
```

```
        plt.show()

        upper =
        ↪  corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
        ↪  k=1).astype(bool))
        to_drop = [column for column in upper.columns if
        ↪  any(upper[column] > 0.9)]
        X = X.drop(to_drop, axis = 1)
```

3. Regularization: I adjusted hyper parameters which controlled how aggressively the model penalizes large weights. This ensured that the model focused on only the strongest patterns.

```
        from tensorflow.keras import Input
        from tensorflow.keras.regularizers import l1_l2
        model = Sequential()
        model.add(Input(shape=(X_train.shape[1],)))
        model.add(Dense(1, activation = 'linear',
              kernel_regularizer=l1_l2(l1=0.01, l2=0.01)))
```

4. Adjust learning rate: I adjusted the learning rate to be smaller (0.0001) so that the model would learn more carefully and avoid mistakes.

```
        from tensorflow.keras.optimizers import Adam
        model.compile(optimizer = Adam(learning_rate = 0.0001),
        ↪  loss =
        'mse', metrics = [rmse, 'mae'])
```

5. Scaling the target variable: I scaled the target variable (y) as well. While this is not a common practice, scaling the target variable is useful when the target variable has a large range or extreme values. By scaling the target variable, everything is numerically stable which helps the model learn faster and more accurately and keeps error low.

```
        y_scaler = StandardScaler()
        y =
        ↪  y_scaler.fit_transform(y.values.reshape(-1,1)).flatten()
```

   (a) However, once you scale the target variable, your predictions will also output scaled values. So in order to convert predicted values into an interpretable form that are comparable to the real world RSRP values given, it is necessary to use the inverse transform function. In this code, I used it in the predictions step, and when creating a visualization, so that my plots reported the transformed values not the scaled values.

```
        #Predictions
        predictions=
        ↪  y_scaler.inverse_transform(predictions.reshape(-1,1)).flatten()
        #Visualizations
        y_pred_unscaled =
        ↪  y_scaler.inverse_transform(y_pred.reshape(-1,1)).flatten()
        y_test_unscaled =
        ↪  y_scaler.inverse_transform(y_pred.reshape(-1,1)).flatten()
```

After these modifications, my RMSE and MAE significantly lowered and my predictions significantly improved. The primary feature used for prediction was Received PDSCH Power (DL). This is the power of the Physical Down link Shared Channel, which is a physical channel used to transfer user data, higher layer control information, and system information from the network to user equipment. I determined this with a simple line of code that output which column was left for use after checking for feature co-linearity and removing highly correlated features.

Metric	**Value**
Root Mean Squared Error (RMSE)	1.31
Mean Absolute Error (MAE)	0.01

Table 6: RMSE and MAE of the revised regression model

Sample	**Actual RSRP (dBm)**	**Predicted RSRP (dBm)**
1	-70.40	-70.47
2	-80.46	-80.40
3	-70.90	-70.97
4	-72.09	-72.14
5	-80.46	-80.40

Table 7: RSRP in Cellular Networks: Predictions vs Actuals

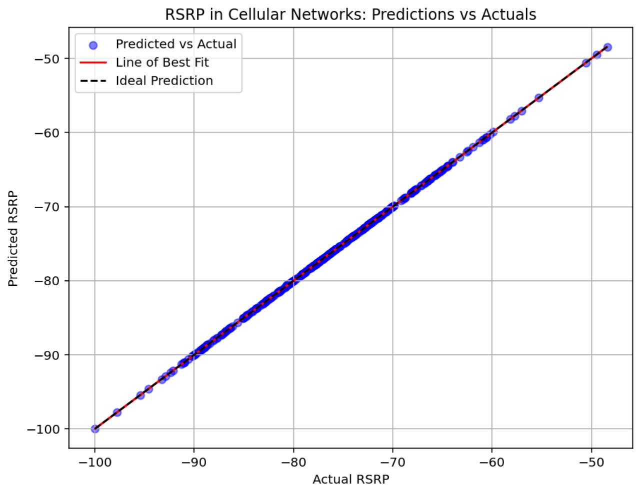

Figure 4: RSRP in Cellular Networks: Predictions vs. Actuals (Revised)

6 Takeaways

Through building a Linear Regression model using an open source dataset, a Linear Regression model using the Signal Strength dataset, and revising the latter, I gained valuable knowledge about machine learning models. I understood the inter workings of a linear regression model and gained a deeper understanding of datasets, specifically how to manipulate them and prepare them for ML models. Additionally, in working with these datasets, I learned how to utilize the sklearn and Keras libraries to source datasets, perform feature extraction, build Linear Regression Models, and create visualizations. Finally, I understood the foundational role platforms such as Spyder and Anaconda Navigator played in building these models.