Week 3: Fitting

Please indicate at the top of your assignment whether or not you used any AI tools, such as MS Copilot. If you did use one of these tools, please provide a very brief explanation alongside each answer for how you confirmed the correctness of your solution.

- Used MS Copilot for quick code solutions in come cases.

In many different cases, we might have a model for how a system works, and want to fit that model to a set of observations.

We're going to investigate the process of fitting using a classic paper that proposed a model for the T cell receptor. Here, the authors develop a mathematical model for how binding occurs and then have observations of how much binding occurs under specific conditions. Identifying whether and how this model fits has led to a better understanding of how our immune system recognizes diseased cells, and how to design T cells that respond to diseases like cancer.

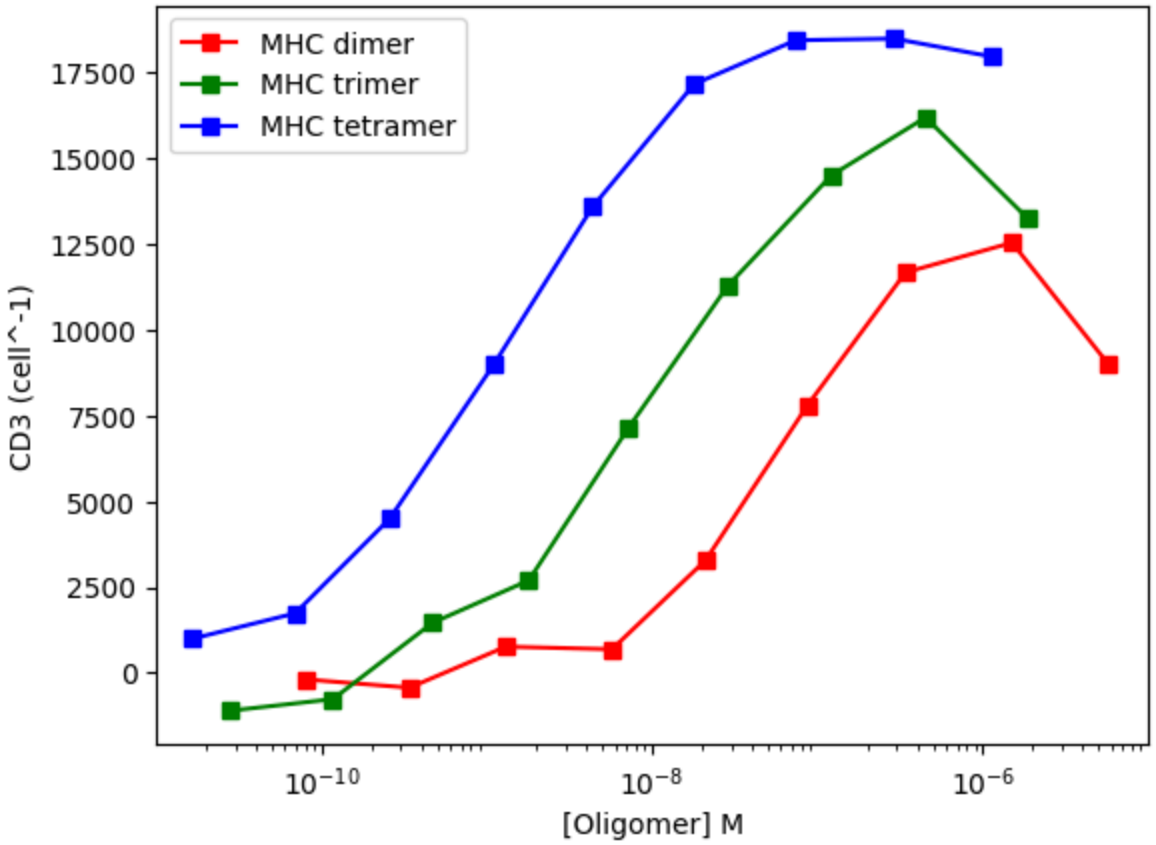
In [44]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import least_squares

np.seterr(over='raise')


def Req_func(Phisum, Rtot, L0, KxStar, f, Ka):
    """ Mass balance. Transformation to account for bounds. """
    Req = Rtot / (1.0 + L0 * f * Ka * (1 + Phisum) ** (f - 1))
    return Phisum - Ka * KxStar * Req


def StoneMod(Rtot: float, Kd: float, v, Kx: float, L0: np.ndarray):
    '''
    Returns the number of mutlivalent ligand bound to a cell with Rtot
    receptors, granted each epitope of the ligand binds to the receptor
    kind in question with dissociation constant Kd and cross-links with
    other receptors with crosslinking constant Kx. All eq derived from Stone et al. (200
    '''
    v = np.int_(v)
    assert L0.shape == v.shape

    if Rtot <= 0.0:
        raise RuntimeError("You input a negative amount of receptor.")

    if Kx <= 0.0:
        raise RuntimeError("You input a negative Kx.")

    if np.amin(L0) <= 0.0:
        raise RuntimeError("You input a negative L0.")

    Ka = 1.0 / Kd
    KxStar = Kx / Ka

    ## Solve Req by calling least_squares
```

```
    lsq = least_squares(Req_func, np.ones_like(L0), jac="cs",
                        max_nfev=5000, xtol=1.0E-10, ftol=1.0E-10, gtol=1.0E-10,
                        args=(Rtot, L0, KxStar, v, Ka))

    if lsq['success'] is False:
        print(lsq)
        raise RuntimeError("Failure in solving for Req. If you see this message contact

    Phisum = lsq.x

    # Calculate L, according to equation 7
    Lbound = L0 / KxStar * ((1 + Phisum) ** v - 1)

    # Calculate Rmulti from equation 5
    Rmulti = L0 / KxStar * v * Phisum * ((1 + Phisum) ** (v - 1) - 1)

    # Calculate Rbound
    Rbnd = L0 / KxStar * v * Phisum * (1 + Phisum) ** (v - 1)

    return Lbound, Rbnd, Rmulti

Xs = np.array([8.1E-11, 3.4E-10, 1.3E-09, 5.7E-09, 2.1E-08, 8.7E-08, 3.4E-07, 1.5E-06, 5
Ys = np.array([-196, -436, 761, 685, 3279, 7802, 11669, 12538, 9012, -1104, -769, 1455,
Vs = np.repeat([2, 3, 4], 9)

print(len(Xs), len(Ys), len(Vs))
```

```
27 27 27
```

(1) We will fit the data contained within Fig. 3B. Plot this data and describe the relationship you see between Kx, Kd, and valency.

In [35]:
```
# Note that Xs are the concentrations from the experiment
# Vs are the complex valencies
# Ys are the measurements of response
# Each of these vectors is the same length, with each position describing an experimenta

def plot_valency_data(Xs_in, Ys_in, Vs_in, format='o-'):
    """If you pass in your real or simulated data, this will plot it for you."""
    colors = ['r', 'g', 'b']

    for valency in range(3):
        plt.semilogx(Xs_in[Vs_in == valency + 2],
                     Ys_in[Vs_in == valency + 2],
                     colors[valency] + format)

    plt.xlim([1E-11, 1E-5])
    plt.xticks([1E-10, 1E-8, 1E-6])
    plt.xlabel('[Oligomer] M')
    plt.ylabel('CD3 (cell^-1)')
    plt.legend(['MHC dimer', 'MHC trimer', 'MHC tetramer'])

# Answer
plot_valency_data( Xs_in=Xs, Ys_in=Ys, Vs_in=Vs, format="s-")
```



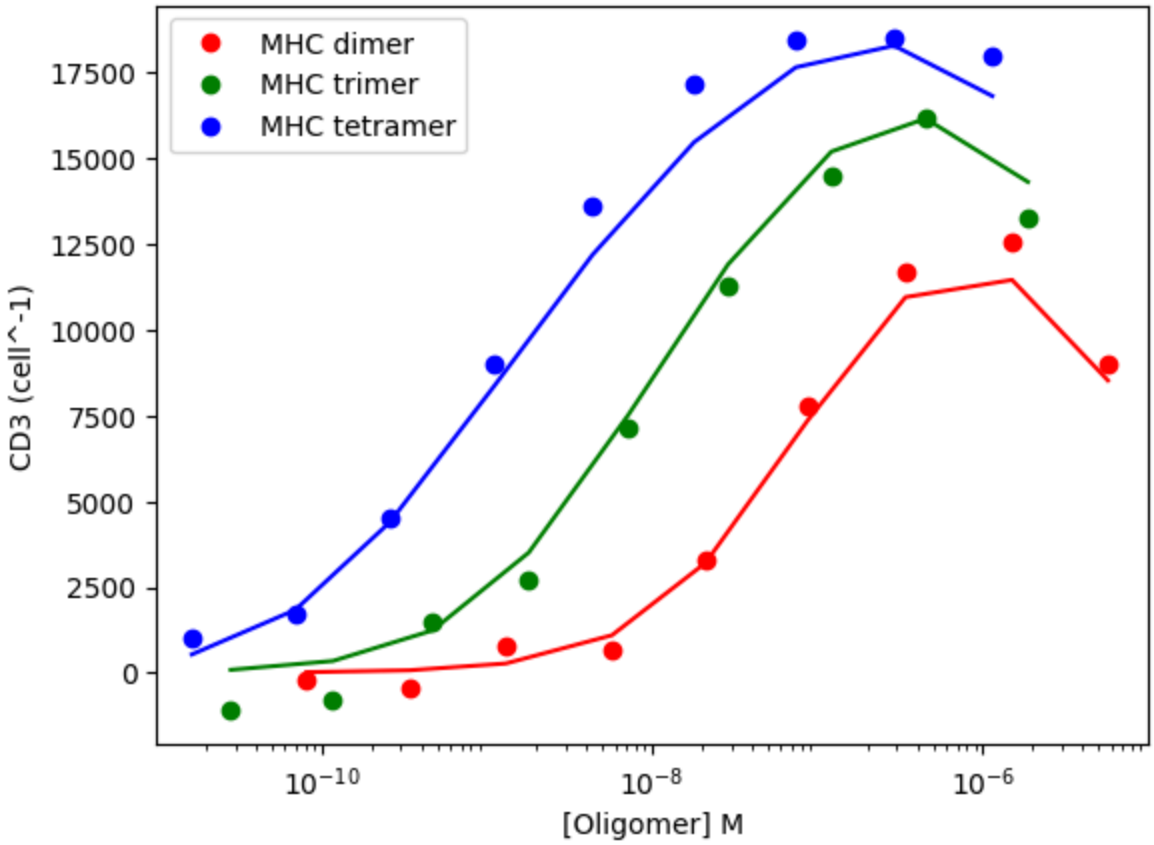
The concentraitons, complex valencies, and the measurement of response show a multi-colinear realtionship where the response is dependent on the concentration and valencies. The plot reveals that once the T-cell receptors achieve saturation, no more oligomers are able to bind. We observe faster saturation with less complex oligomers (e.g., MHC dimer). With increased valency we observe binding of more complex oligomers while requiring lower concentrations of oligomers for efficient binding and greater receptor multimerization. Overall, increasing valency seems to shifts peaks to the left. High Kx (higher avidity) can lead to an reduction in the effective Kd (dissociation constant) as the cross-linked complexes provide more stablity.

(2) First, to do so, we'll need a function that takes the model predictions, scales them to the units of the actual measurements, and finds the predictions for each condition. Define a scaling parameter and a function that takes it along with the other parameters to make predictions about the experiment.

Use the fit parameters shown in Table 1 (row 2) and overlay with the measurements to ensure your function is working. (Scale = 1 for now.)

In [36]:
```
# Here is a scaling function
# Note that the unknown parameters are passed in as an array
# This will make working with least_squares() easier later
# parameters = [scale, Kd (M), Kx (cell)]
def scale_StoneMod(parameters, Vs, Xs):
    Rtot = 24000.0 # cell^-1
    Lbound, Rbnd, Rmulti = StoneMod(Rtot, parameters[1], Vs, parameters[2], Xs)
    return parameters[0] * Rmulti

parameters = [1, 1.7E-6, 3.15E-4] # For CD3 @ 27 hours
prediction = scale_StoneMod(parameters=parameters, Vs=Vs, Xs=Xs)
```

```
# Note that Xs are the concentrations from the experiment
# Vs are the complex valencies
# Ys are the measurements of response
plot_valency_data( Xs_in=Xs, Ys_in=Ys, Vs_in=Vs, format='o') # Actual
plot_valency_data( Xs_in=Xs, Ys_in=prediction, Vs_in=Vs, format='-') # Prediction
```

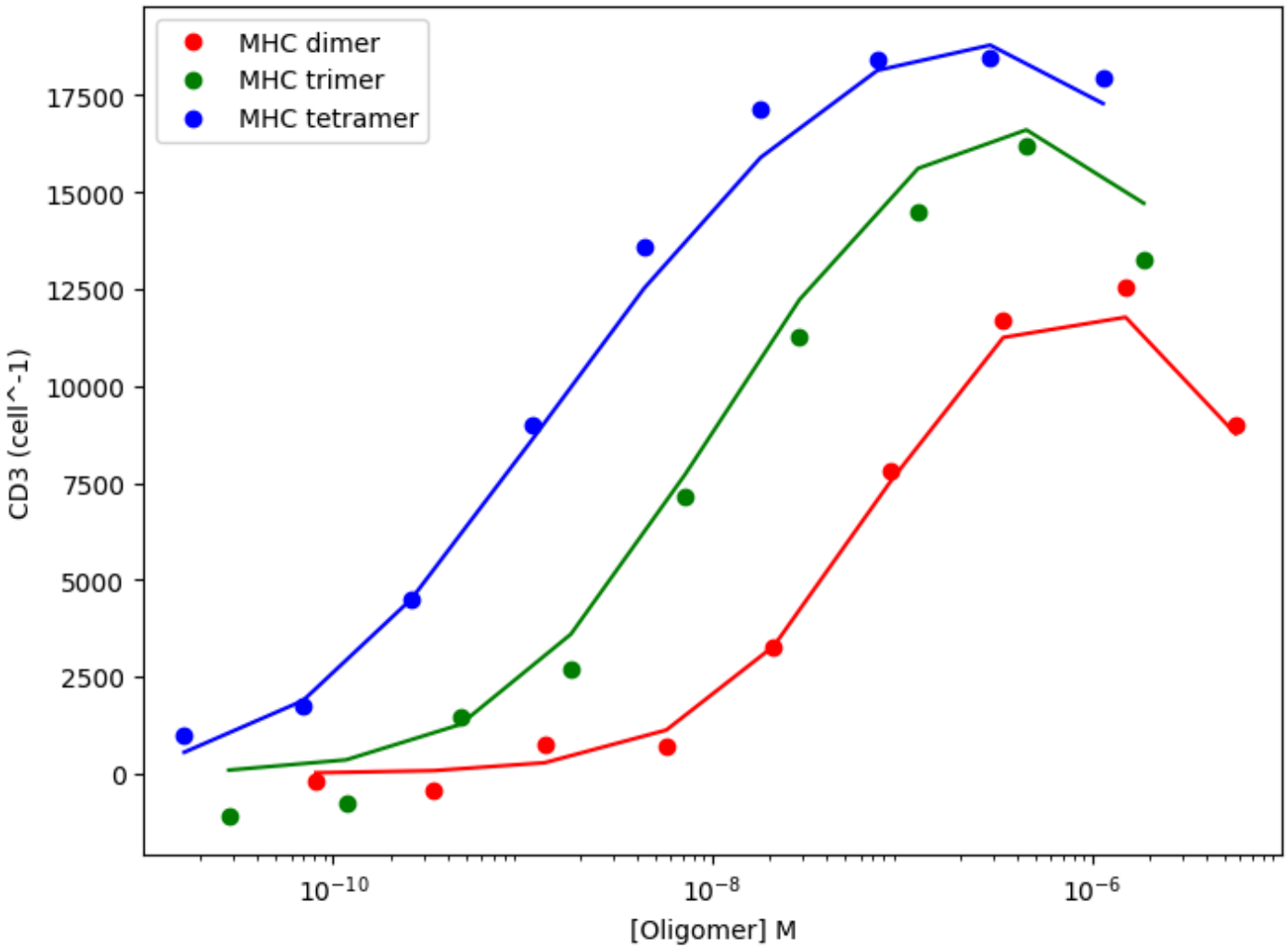


(3) Now use `scipy.optimize.least_squares` to find the least squares solution.

You should use information from the paper to start at a good guess for the parameter values.

In [37]:
```
# You will need a function that returns the residuals:
def residuals(parameters, Vs_in, Xs_in, Ys_in):
    return Ys_in - scale_StoneMod(parameters, Vs_in, Xs_in)

# Note that the data (Xs, Ys) are not unknowns, so they should be passed in using the ar
# With this function, your least squares call should look something like this:

# lsq_solution = least_squares(residuals, params_guess, args=(Vs, Xs, Ys))


params_guess = [1, 1.7E-6, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
lsq_solution = least_squares(residuals, x0=params_guess, args=(Vs, Xs, Ys))

lsq_scale = lsq_solution.x[0]
lsq_kd = lsq_solution.x[1] # dissociation constant
lsq_kx = lsq_solution.x[2] # cross-linking constant

optimized_parameters = [lsq_scale, lsq_kd, lsq_kx]
prediction = scale_StoneMod(optimized_parameters, Vs, Xs)


plt.figure(figsize=(8, 6))
```

```
plot_valency_data(Xs, Ys, Vs, 'o')
plot_valency_data(Xs, prediction, Vs, '-')
```

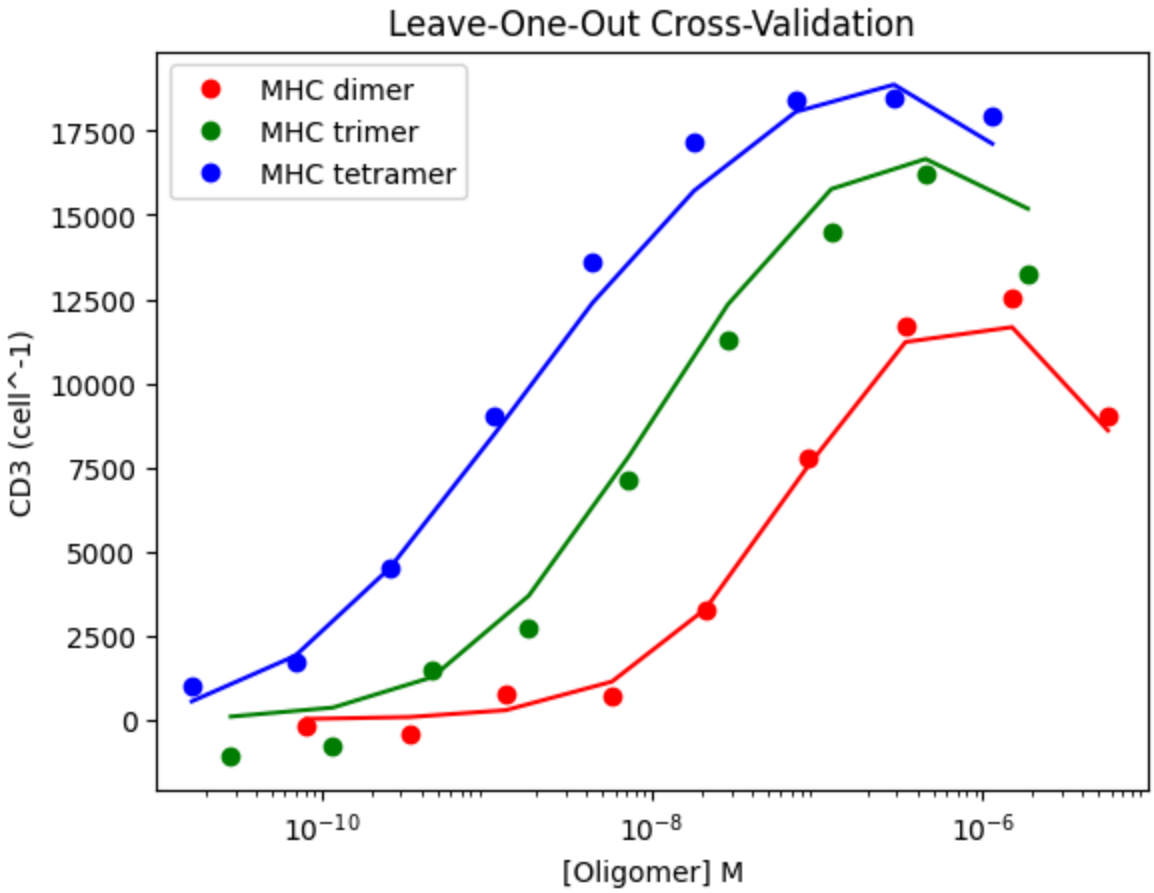


4) Using leave-one-out crossvalidation, does this model predict the data? Plot the measured vs. predicted data.

In [38]:
```
# scikit-learn provides a great interface for getting indices that help you split the da
from sklearn.model_selection import LeaveOneOut
from sklearn.metrics import root_mean_squared_error


params_guess = [1, 1.7E-6, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
predictions = np.zeros_like(Ys)
X_cv = np.zeros_like(Xs)
V_cv = np.zeros_like(Vs)
errors = np.zeros_like(Ys)
rmse = np.zeros_like(Ys)

# CV (optimizing for parameters and predicting with test set)
for train_idx, test_idx in LeaveOneOut().split(Vs):

    Xs_train, Xs_test = Xs[train_idx], Xs[test_idx]
    Ys_train, Ys_test = Ys[train_idx], Ys[test_idx]
    Vs_train, Vs_test = Vs[train_idx], Vs[test_idx]

    lsq_solution = least_squares(residuals, x0=params_guess, args=(Vs_train, Xs_train, Y
```

```
    lsq_scale = lsq_solution.x[0]
    lsq_kd = lsq_solution.x[1]
    lsq_kx = lsq_solution.x[2]

    optimized_parameters = [lsq_scale, lsq_kd, lsq_kx]
    prediction = scale_StoneMod(optimized_parameters, Vs_test, Xs_test)

    predictions[test_idx] = prediction[0]
    X_cv[test_idx] = Xs_test[0]
    V_cv[test_idx] = Vs_test[0]

    errors[test_idx] = (Ys_test - prediction)
    rmse[test_idx] = root_mean_squared_error(Ys_test, prediction)


# Note that Xs are the concentrations from the experiment
# Vs are the complex valencies
# Ys are the measurements of response

print ("RMSE", np.mean(rmse))
plot_valency_data(Xs, Ys, Vs, 'o')
plot_valency_data(X_cv, predictions, V_cv, '-')
plt.title('Leave-One-Out Cross-Validation')
plt.legend(['MHC dimer', 'MHC trimer', 'MHC tetramer'])
plt.show()
```

```
RMSE 665.3703703703703
```



The model is able to capture the general trend in the data, but it is not able to capture the exact values of the data. This is likely due to the fact that the model is able to generalize well. There are many factors that are not accounted for in the model. Additionally, the model assumes that the binding of the

ligand to the receptor is a simple equilibrium process, which may not be true due to difusion around the biological environment. With our current model, we achieve a RMSE value of 665.37.

(5) Using bootstrap estimation, plot the confidence interval of the model predictions along with the data points.

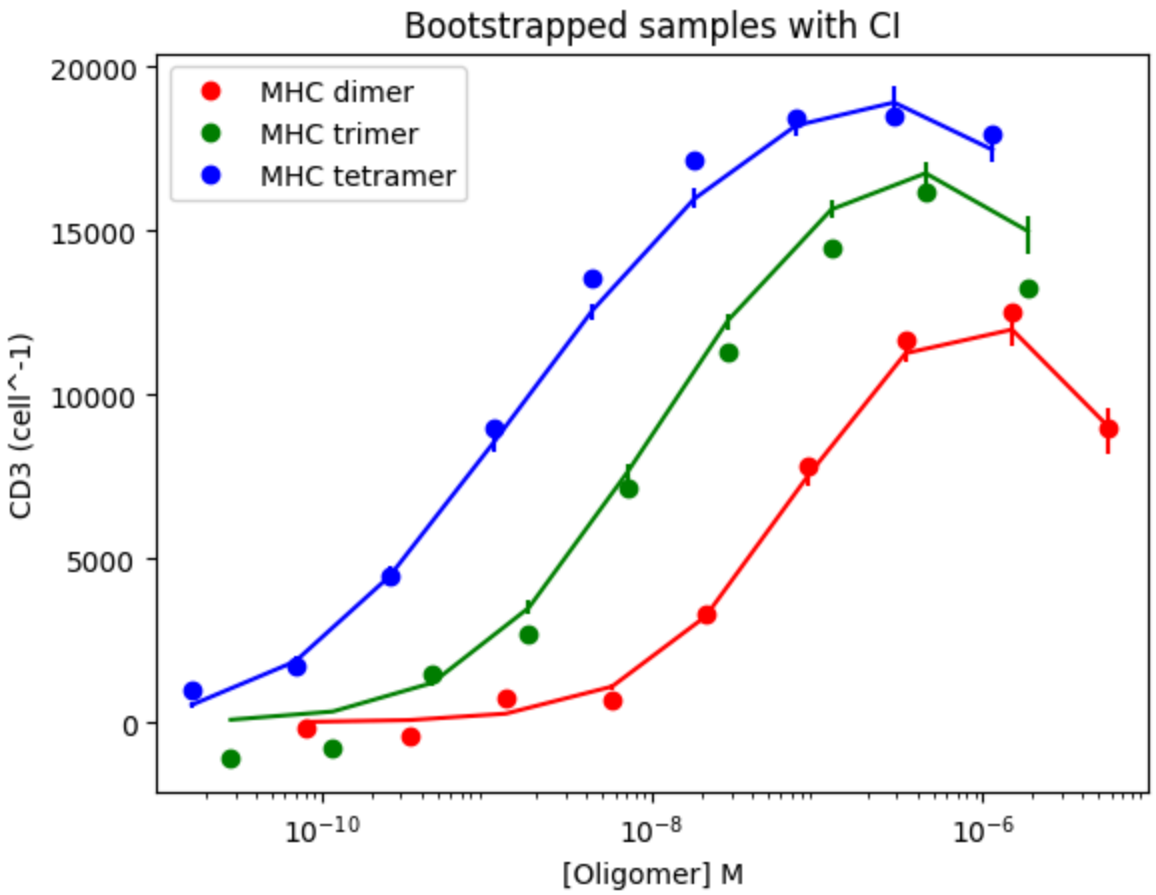
"Confidence interval" does not have a precise definition. For example, you could show the interval over which 50% of the bootstrap samples fall (25th to 75th quantile).

In [39]:
```
# sklearn has a very useful resample function
from sklearn.utils import resample
import pandas as pd
import numpy as np

params_guess = [1, 1.7E-6, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
n_bootstraps = 100
samples = len(Xs)//2
output_data = []

# Bootstrapping (only optimizing for paramters)
for idx in range(n_bootstraps):
    Vs_resamp, Xs_resamp, Ys_resamp = resample(Vs, Xs, Ys, replace=True, n_samples=sampl

    lsq_solution = least_squares(residuals, x0=params_guess, args=(Vs_resamp, Xs_resamp,
    lsq_scale = lsq_solution.x[0]
    lsq_kd = lsq_solution.x[1]
    lsq_kx = lsq_solution.x[2]

    # Note that Xs are the concentrations from the experiment
    # Vs are the complex valencies
    # Ys are the measurements of response

    optimized_parameters = [lsq_scale, lsq_kd, lsq_kx]
    prediction = scale_StoneMod(optimized_parameters, Vs, Xs)

    q25 = np.percentile(prediction, 25)
    median = np.median(prediction)
    q75 = np.percentile(prediction, 75)
    error_bar = np.array([median - q25, q75 - median])
    variance = np.var(prediction)

    output_data.append({
        'scale': lsq_scale,
        'kd': lsq_kd,
        'kx': lsq_kx,
        'predictions': prediction,
        'q25': q25,
        'median': median,
        'q75': q75,
        'error_bars': error_bar,
        'variance': variance
    })

output_df = pd.DataFrame(output_data)
```

In [40]:
```
median = np.percentile(np.array(output_df['predictions'].tolist()), 50, axis=0)
lower_ci = np.percentile(np.array(output_df['predictions'].tolist()), 25, axis=0)
```

```
upper_ci = np.percentile(np.array(output_df['predictions'].tolist()), 75, axis=0)
error_bars = pd.DataFrame([median - lower_ci, upper_ci - median])

plot_valency_data(Xs, Ys, Vs, 'o')
colors = ['r', 'g', 'b']

for valency in range(3): # 3 valencies
    plt.errorbar(Xs[9*valency:9*(valency+1)], median[9*valency:9*(valency+1)],
                 yerr=np.array(error_bars.iloc[:, 9*valency:9*(valency+1)]),
                 color=colors[valency])

plt.title('Bootstrapped samples with CI')
```

Out[40]: Text(0.5, 1.0, 'Bootstrapped samples with CI')



The confidene intervals on the plots display the 25th and 75th percentile for each bootstrapped point. The boostrap trainings randomly samples one half of the traning data with replacement to train and predict parameters. The optimized parametrers through bootstrapping were then used to build the model and make predictions.

(6) Use bootstrap sampling to show the uncertainty in the model prediction for the amount of `Lbnd` for 1 nM of MHC-peptide complex of valency 4. Ensure that you show correct units.

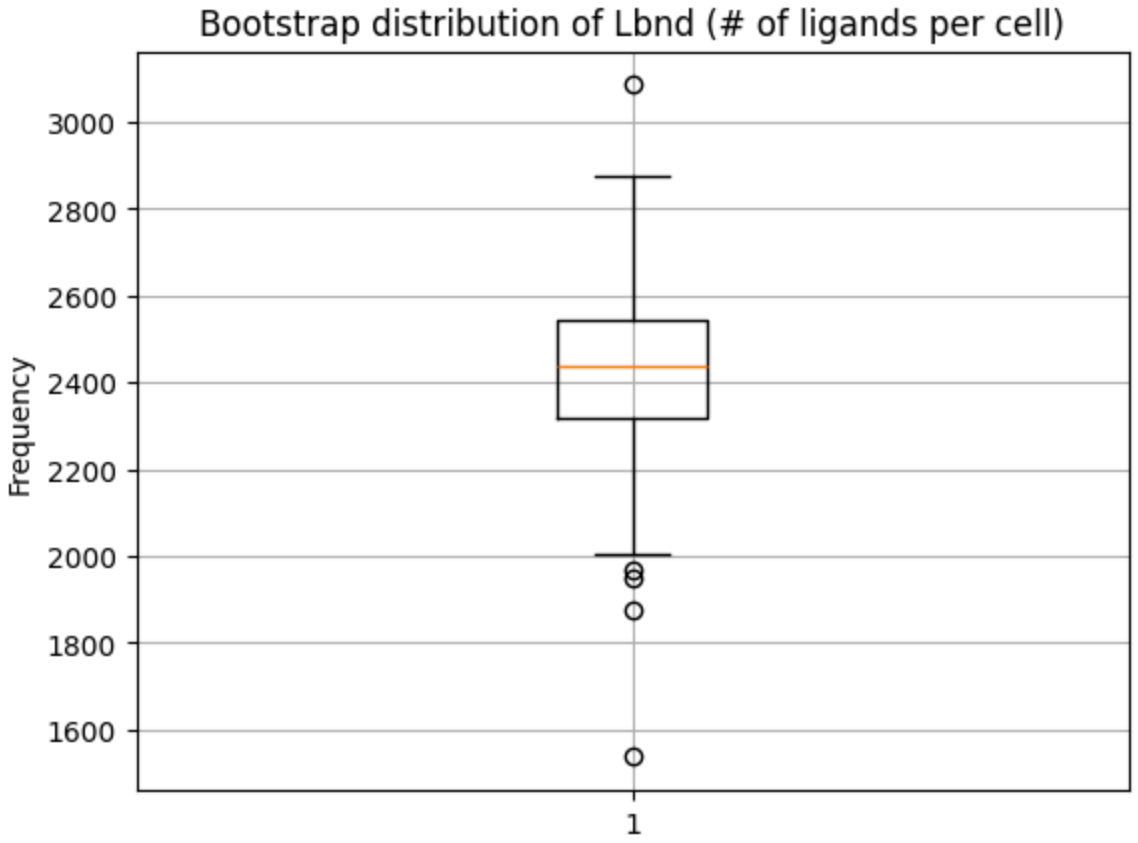
In [41]:
```
from sklearn.utils import resample
import pandas as pd
import numpy as np

# Parameters
params_guess = [1, 1.7E-6, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
```

```
n_bootstraps = 100
samples = len(Xs)//2
output_data = []
Rtot = 24000.0
Kd = 1.7E-6
valency = 4
Kx = 3.15E-4
L0_conc = np.array(1E-9)

# Bootstrapping (only optimizing for paramters)
for idx in range(n_bootstraps):
    Vs_resamp, Xs_resamp, Ys_resamp = resample(Vs, Xs, Ys, replace=True, n_samples=sampl

    lsq_solution = least_squares(residuals, x0=params_guess, args=(Vs_resamp, Xs_resamp,
    lsq_scale = lsq_solution.x[0]
    lsq_kd = lsq_solution.x[1]
    lsq_kx = lsq_solution.x[2]

    # Note that Xs are the concentrations from the experiment
    # Vs are the complex valencies
    # Ys are the measurements of response

    optimized_parameters = [lsq_scale, lsq_kd, lsq_kx]
    Lbound, Rbnd, Rmulti = StoneMod(Rtot=Rtot, Kd=optimized_parameters[1], v=valency, Kx

    output_data.append({
        'Lbnd (# of ligands per cell)': Lbound[0],
        'Rbnd (# of receptors bound)': Rbnd[0],
        'Rmulti (# multimerized receptors)': Rmulti[0]
    })


output_df = pd.DataFrame(output_data)
```

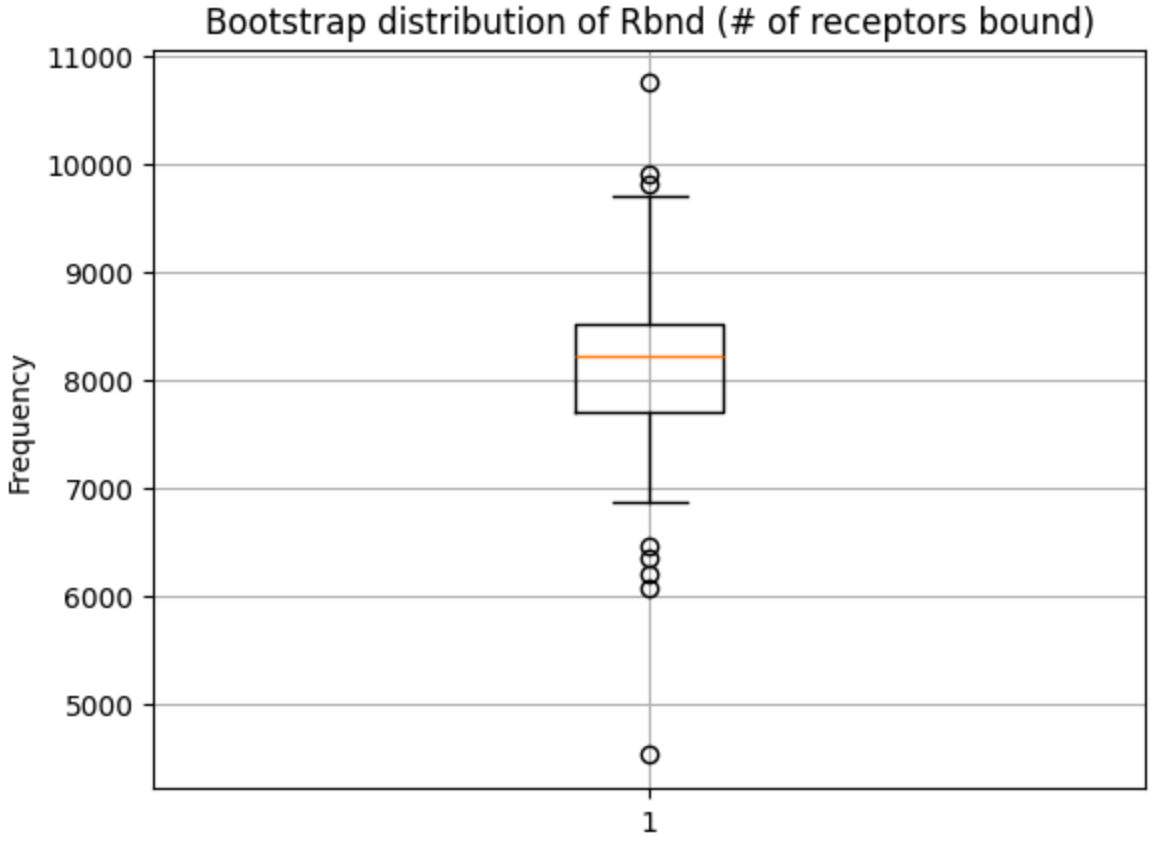
In [43]:

```
# Function to create histograms for each metric
def plot_metric_histogram(metric_name, metric_label, xlabel):

    metric_data = np.array(output_df[metric_name])
    median = np.percentile(metric_data, 50)
    lower_ci = np.percentile(metric_data, 25)
    upper_ci = np.percentile(metric_data, 75)

    print(f"Median {metric_label} for 1 nM MHC-peptide complex: {median:.4e}")
    print(f"25th Percentile (Lower CI): {lower_ci:.4e}")
    print(f"75th Percentile (Upper CI): {upper_ci:.4e}")

    plt.boxplot(metric_data)
    plt.ylabel('Frequency')
    plt.title(f'Bootstrap distribution of {metric_label}')
    plt.xlabel('')
    plt.grid(True)
    plt.show()

# Plotting each metric with the same style
plot_metric_histogram('Lbnd (# of ligands per cell)', 'Lbnd (# of ligands per cell)', ''
plot_metric_histogram('Rbnd (# of receptors bound)', 'Rbnd (# of receptors bound)', '')
plot_metric_histogram('Rmulti (# multimerized receptors)', 'Rmulti (# multimerized recep
```

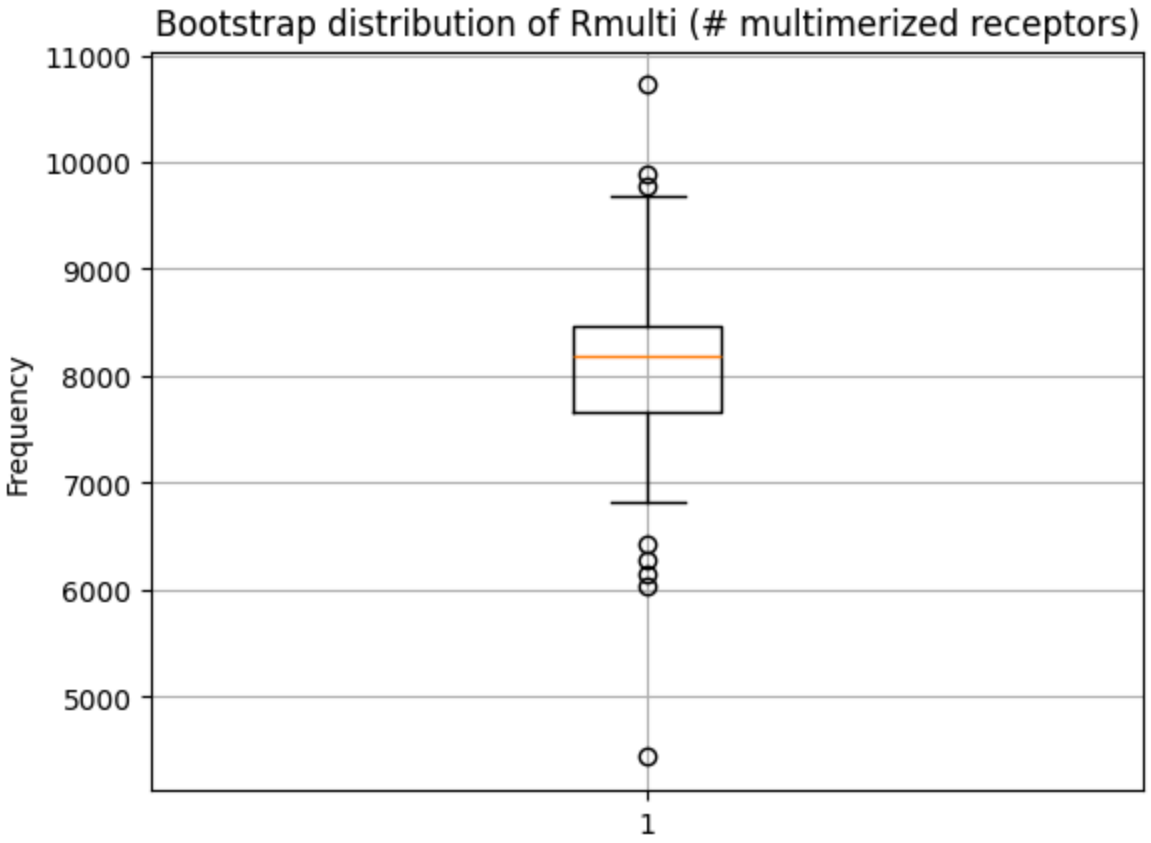
Median Lbnd (# of ligands per cell) for 1 nM MHC–peptide complex: 2.4377e+03
25th Percentile (Lower CI): 2.3197e+03
75th Percentile (Upper CI): 2.5428e+03



Median Rbnd (# of receptors bound) for 1 nM MHC–peptide complex: 8.2144e+03
25th Percentile (Lower CI): 7.7075e+03
75th Percentile (Upper CI): 8.5109e+03



```
Median Rmulti (# multimerized receptors) for 1 nM MHC-peptide complex: 8.1858e+03
25th Percentile (Lower CI): 7.6641e+03
75th Percentile (Upper CI): 8.4762e+03
```



(7) *Generally*, how would you expect the cross-validation and bootstrap results to change if you had fewer data points?

Cross-validation

- With fewer points we will have a smaller training set which can lead to less generalizability because of the less representative nature of the data.
- There is a higher change of overefitting with less data.
- Predicion error is overestimated.
- Higher folds would lead to increased variance, and reducing the number of folds would introduce more bias reducing the predicion accuracy.

Bootstratpping

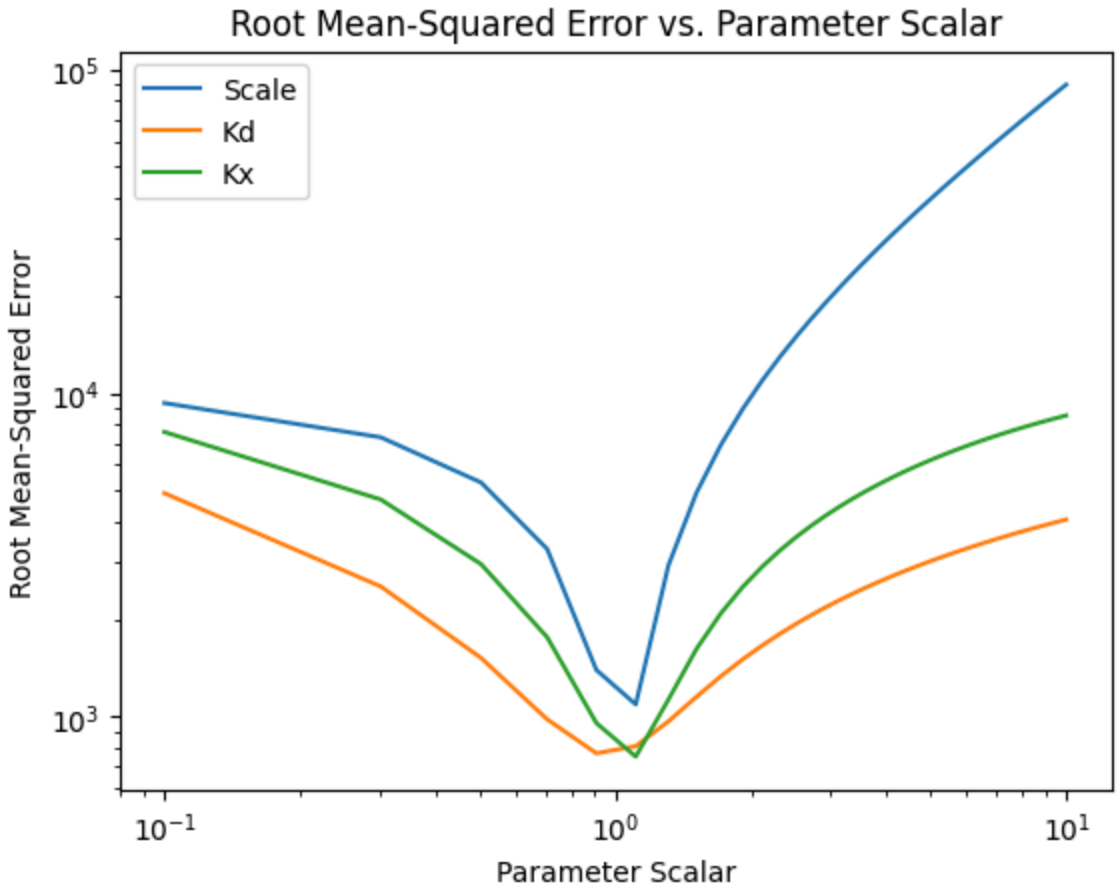
- We will observe larger variance with each point estimated.
- Less data within each bootstapping fold would not be helpful in fitting the data.
- Overall, will results in greater variance in our predictions.
- Prediction error is underestimated.

(8) Now, we will perform a local sensitivity analysis to look at the dependence of the model results on each parameter. Vary each parameter up and down relative to its optimum by 10-fold **while holding the others constant**, and plot the sum of squared error. Which parameter affects the error the most? Which one the least?

You should get a plot that roughly looks like a "U" for each parameter. Use a log-log plot to show the variation the best (`plt.loglog`).

In [23]:
```
from sklearn.metrics import root_mean_squared_error

param_set = np.linspace(0.1, 10)
rmse_scale = []
rmse_kd = []
rmse_kx = []

for param in param_set:

    params_mod1 = [1*param, 1.7E-6, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
    mod_scale = scale_StoneMod(params_mod1, Vs, Xs)
    rmse_scale.append(root_mean_squared_error(mod_scale, Ys))

    params_mod2 = [1, 1.7E-6*param, 3.15E-4] # parameters = [scale, Kd (M), Kx (cell)]
    mod_kd = scale_StoneMod(params_mod2, Vs, Xs)
    rmse_kd.append(root_mean_squared_error(mod_kd, Ys))

    params_mod3 = [1, 1.7E-6, 3.15E-4*param] # parameters = [scale, Kd (M), Kx (cell)]
    mod_kx = scale_StoneMod(params_mod3, Vs, Xs)
    rmse_kx.append(root_mean_squared_error(mod_kx, Ys))
```

In [24]:
```
plt.figure()
plt.loglog(param_set, rmse_scale)
plt.loglog(param_set, rmse_kd)
plt.loglog(param_set, rmse_kx)
plt.ylabel('Root Mean-Squared Error')
plt.xlabel('Parameter Scalar')
plt.title('Root Mean-Squared Error vs. Parameter Scalar')
plt.legend(['Scale', 'Kd', 'Kx'])
plt.yscale('log')
plt.show()
```



The scale paramter seems to have the most effect on fit as we can see significant changes to the RMSE as the parameter changes. Kd, the dissociation constant, has the least influence on fitting which shows that changes to this paramter affect the RMSE the least out of the three parameters.