

# Predicting NHL Clutch Goalscorers

December 28, 2025

## 1 Predicting NHL Clutch Goalscorers

This project applies machine learning techniques to identify and predict NHL forwards who excel in “clutch” situations (close, tied, and overtime games). The goal is not only to measure clutch performance but also to model expected clutch scoring given a player’s underlying metrics and understand the reasoning behind the predictions.

The final model has been deployed to a [Streamlit Dashboard](#) that is updated at 9:00 a.m. EST daily.

```
[3]: import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

from sklearn.exceptions import FitFailedWarning
warnings.filterwarnings("ignore", category=FitFailedWarning)

import time
import math
import json
import requests
import functools as ft
import scipy.stats as stats

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm

import xgboost as xgb
from xgboost import XGBClassifier, plot_importance

from sklearn.model_selection import train_test_split, StratifiedKFold,
    ↪cross_validate, learning_curve
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↪f1_score, mean_squared_error, mean_absolute_error, r2_score,
    ↪median_absolute_error, PrecisionRecallDisplay, make_scorer
from sklearn.linear_model import Ridge, RidgeCV, LinearRegression
```

```

from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_sample_weight
from sklearn.decomposition import PCA
from sklearn.utils import resample

import shap

from skopt import BayesSearchCV
from skopt.space import Integer, Real, Categorical

import joblib

```

### 1.0.1 NHL API

```

[5]: all_seasons = []

for season in range(2021, 2024):
    summary_url = f"https://api.nhle.com/stats/rest/en/skater/summary?
    ↪limit=-1&cayenneExp=seasonId={season}-{season+1}%20and%20gameTypeId=2"

    try:
        summary_resp = requests.get(summary_url)
        summary_resp.raise_for_status()
        summary_json = summary_resp.json()

        if summary_json['data']:
            df_summary = pd.DataFrame(summary_json['data'])
            all_seasons.append(df_summary)
            df_summary['season'] = f"{season}-{season + 1}"
            print(f"Successfully fetched data for season {season}-{season+1}")
        else:
            print(f"No data returned for season {season}-{season + 1}")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching data for season {season}-{season + 1}: {e}")

if all_seasons:
    nhl_api_df = pd.concat(all_seasons, ignore_index=True)
    nhl_api_df = nhl_api_df.groupby('playerId').agg({
        'playerId': 'first',
        'skaterFullName': 'first',
        'positionCode': 'first',
        'gamesPlayed': 'sum',
        'assists': 'sum',
        'otGoals': 'sum',
        'timeOnIcePerGame': 'mean'
    }).reset_index(drop = True)

```

```
print(nhl_api_df)
```

Successfully fetched data for season 2021-2022

Successfully fetched data for season 2022-2023

Successfully fetched data for season 2023-2024

	playerId	skaterFullName	positionCode	gamesPlayed	assists	otGoals	\
0	8465009	Zdeno Chara	D	72	12	0	
1	8466138	Joe Thornton	C	34	5	0	
2	8469455	Jason Spezza	C	71	13	0	
3	8470281	Duncan Keith	D	64	20	0	
4	8470595	Eric Staal	C	72	15	0	
...	...	...	...	...	...	...	
1250	8484314	Jiri Smejkal	L	20	1	0	
1251	8484321	Nikolas Matinpalo	D	4	0	0	
1252	8484325	Waltteri Merela	C	19	0	0	
1253	8484326	Patrik Koch	D	1	0	0	
1254	8484911	Collin Graf	R	7	2	0	

	timeOnIcePerGame
0	1123.9027
1	666.3529
2	644.7605
3	1183.6093
4	854.2222
...	...
1250	568.7000
1251	420.2500
1252	588.9473
1253	560.0000
1254	995.7142

[1255 rows x 7 columns]

## 1.0.2 Cleaning the NHL API Data

- Only forwards are included since defensemen score at different rates.
- Players must have appeared in at least 60 games across the three seasons (approximately 20 games each season). This ensured that there was a sufficient sample size for each player.

```
[7]: nhl_api_df = nhl_api_df.loc[(nhl_api_df['positionCode'] != 'D') &
    ↪(nhl_api_df['gamesPlayed'] >= 60)]
nhl_api_df = nhl_api_df.reset_index(drop = True)

rename_columns = {
    'otGoals': 'ot_goals',
    'skaterFullName': 'Player',
    'timeOnIcePerGame': 'time_on_ice_per_game'
}
```

```
nhl_api_df.rename(columns = rename_columns, inplace = True)
```

### 1.0.3 Scraping Data from Natural Stat Trick

```
[9]: start_season = "20212022"
end_season = "20232024"
goals_up_one_url = f"https://www.naturalstattrick.com/playerteams.php?
↳fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=u1&stdoi=std&rate=n
goals_down_one_url = f"https://www.naturalstattrick.com/playerteams.php?
↳fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=d1&stdoi=std&rate=n
tied_url = f"https://www.naturalstattrick.com/playerteams.php?
↳fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=tied&stdoi=std&rate=
total_url = f"https://www.naturalstattrick.com/playerteams.php?
↳fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=all&stdoi=std&rate=
on_ice_url = f"https://www.naturalstattrick.com/playerteams.php?
↳fromseason={start_season}&thruseason={end_season}&stype=2&sit=5v5&score=all&stdoi=oi&rate=n
```

```
[10]: urls = {
    "goals_up_one": (goals_up_one_url, 'goals_up_by_one'),
    "goals_down_one": (goals_down_one_url, 'goals_down_by_one'),
    "tied": (tied_url, 'goals_when_tied'),
    "total": (total_url, 'total_goals'),
    "on_ice": (on_ice_url, '')
}

dataframes = {}

for name, (url, new_column_name) in urls.items():
    df = pd.read_html(url, header=0, index_col=0, na_values=["-"])[0]
    df.rename(columns={'Goals': new_column_name}, inplace=True)
    dataframes[name] = df

goals_up_one_df = dataframes["goals_up_one"]
goals_down_one_df = dataframes["goals_down_one"]
goals_tied_df = dataframes["tied"]
total_df = dataframes["total"]
on_ice_df = dataframes["on_ice"]
on_ice_df.columns = on_ice_df.columns.str.replace('\xa0', ' ')
```

### 1.0.4 Cleaning Data from Natural Stat Trick

Similar to the NHL API data, only players who have played at least 60 games are included. The dataframes have already been filtered for forwards through the URLs.

```
[12]: goals_up_one_df = goals_up_one_df[['Player', 'GP', 'goals_up_by_one']]
goals_down_one_df = goals_down_one_df[['Player', 'goals_down_by_one']]
```

```

goals_tied_df = goals_tied_df[['Player', 'goals_when_tied']]
total_df = total_df[['Player', 'total_goals', 'Shots', 'ixG', 'iFF', 'iSCF',
↳ 'iHDCF', 'Rebounds Created', 'iCF']]
on_ice_df = on_ice_df[['Player', 'Off. Zone Starts', 'On The Fly Starts']]

dfs_natural_stat = [goals_up_one_df, goals_down_one_df, goals_tied_df,
↳ total_df, on_ice_df]

merged_natural_stat = ft.reduce(lambda left, right: pd.merge(left, right,
↳ on='Player'), dfs_natural_stat)
merged_natural_stat = merged_natural_stat.loc[merged_natural_stat['GP'] >= 60]

rename_columns = {
    'Shots': 'shots',
    'Rebounds Created': 'rebounds_created',
    'Off. Zone Starts': 'off_zone_starts',
    'On The Fly Starts': 'on_the_fly_starts'
}

merged_natural_stat.rename(columns = rename_columns, inplace=True)

```

### 1.0.5 Standardize Player Names

Some players from Natural Stat Trick have different names compared to the NHL API. It is important to use standard names in both dataframes before merging them.

```

[14]: natural_stat_names = ["Pat Maroon", "Alex Kerfoot", "Nicholas Paul", "Zach
↳ Sanford", "Alex Wennberg", "Mitchell Marner", "Max Comtois", "Alexei
↳ Toropchenko", "Cameron Atkinson", "Thomas Novak", "Zack Bolduc", "Frederic
↳ Gaudreau"]
nhl_names = ["Patrick Maroon", "Alexander Kerfoot", "Nick Paul", "Zachary
↳ Sanford", "Alexander Wennberg", "Mitch Marner", "Maxime Comtois", "Alexey
↳ Toropchenko", "Cam Atkinson", "Tommy Novak", "Zachary Bolduc", "Freddy
↳ Gaudreau"]
merged_natural_stat = merged_natural_stat.replace(natural_stat_names, nhl_names)

```

### 1.0.6 Merging the Data

The dataframes containing the information from the NHL API and Natural Stat Trick are merged.

```

[16]: merged_clutch_goals = nhl_api_df.merge(merged_natural_stat, on = 'Player', how
↳ = 'left')
merged_clutch_goals = merged_clutch_goals.dropna()

```

### 1.0.7 Changing Columns

Compute per game stats to accurately compare players.

```
[18]: merged_clutch_goals.drop(columns = 'GP', axis = 1, inplace = True)
columns = ['ot_goals', 'assists', 'goals_up_by_one', 'goals_down_by_one',
↳ 'goals_when_tied', 'shots', 'ixG', 'iFF', 'iSCF', 'iHDCF', 'iCF',
↳ 'rebounds_created', 'off_zone_starts', 'on_the_fly_starts']
for column in columns:
    per_game_string = f"{column}_per_game"
    merged_clutch_goals[per_game_string] = merged_clutch_goals[column] /
↳ merged_clutch_goals['gamesPlayed']
```

### 1.0.8 Clutch Score

After cleaning the data, we can now compute a weighted clutch score for each player. - Goals scored when tied and down by one are given the most weighting since these are the most representative of high-pressure situations. - Goals scored when up by one are still close situations but may not be as “clutch” compared to goals scored when tied and down by one. - OT goals are also given a smaller weight, since they occur infrequently compared to other goals. They are also only scored during 3v3 play, which differs from regular 5v5.

```
[20]: merged_clutch_goals['clutch_score'] = (
    0.45 * merged_clutch_goals['goals_down_by_one_per_game'] +
    0.35 * merged_clutch_goals['goals_when_tied_per_game'] +
    0.2 * merged_clutch_goals['ot_goals_per_game']
)
```

### 1.0.9 Rankings Players Based on their Clutch Score

All scores are multiplied by 100 to make them more interpretable. The scores are then ranked and the top 20 players are shown below.

```
[22]: merged_clutch_goals['clutch_score'] *= 100
merged_clutch_goals['clutch_score_rank'] = merged_clutch_goals['clutch_score'].
↳ rank(ascending = False, method = 'min')
merged_clutch_goals['clutch_score'] = merged_clutch_goals['clutch_score'].
↳ apply(lambda x: round(x, 2))
merged_clutch_goals.sort_values('clutch_score_rank', inplace = True)
merged_clutch_goals[['Player', 'clutch_score', 'clutch_score_rank']].head(20)
```

```
[22]:
```

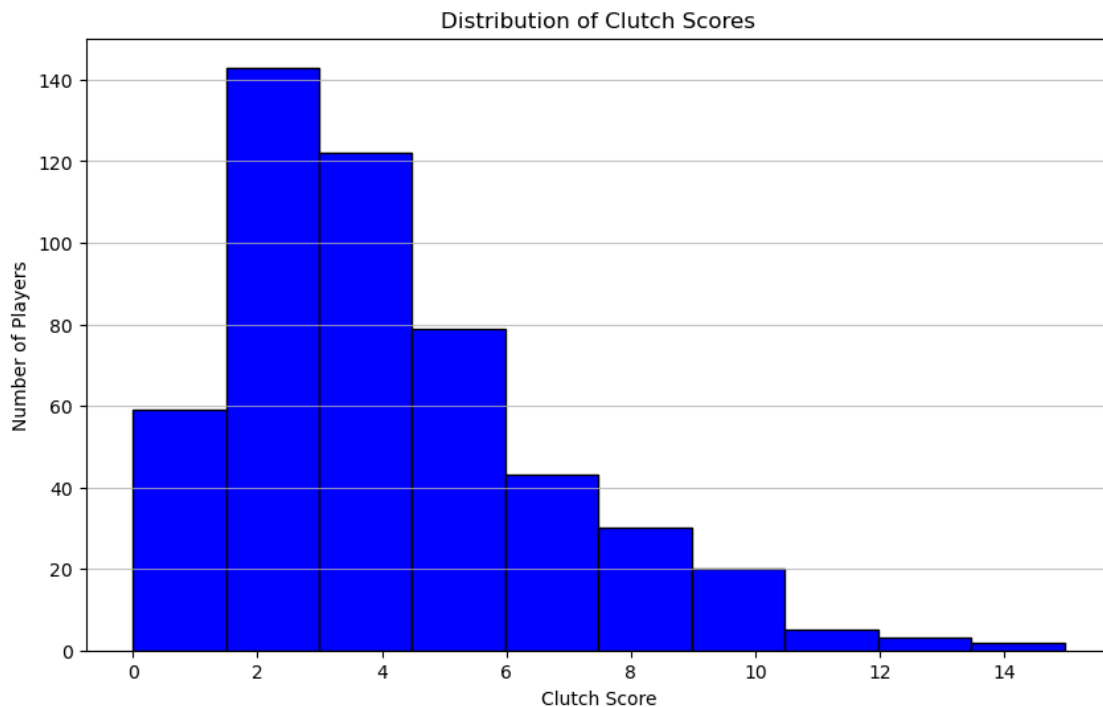
	Player	clutch_score	clutch_score_rank
318	Auston Matthews	14.96	1.0
236	David Pastrnak	13.50	2.0
304	Kirill Kaprizov	13.25	3.0
222	Leon Draisaitl	12.26	4.0
267	Connor McDavid	12.16	5.0
152	Filip Forsberg	11.44	6.0
453	Jack Hughes	11.01	7.0
245	Brayden Point	10.81	8.0
346	Tage Thompson	10.64	9.0
50	Steven Stamkos	10.52	10.0

224	William Nylander	10.08	11.0
270	Timo Meier	10.00	12.0
390	Josh Norris	10.00	13.0
229	Dylan Larkin	9.93	14.0
264	Kyle Connor	9.89	15.0
283	Roope Hintz	9.83	16.0
271	Mikko Rantanen	9.79	17.0
201	Nathan MacKinnon	9.75	18.0
221	Sam Reinhart	9.69	19.0
385	Jason Robertson	9.66	20.0

### 1.0.10 Distribution of Clutch Scores

As shown by the histogram below, the data for clutch scores is right skewed. Most players have a below average clutch score and there are a small number of elite players

```
[24]: plt.figure(figsize=(10, 6))
plt.hist(merged_clutch_goals['clutch_score'], color='blue', edgecolor='black')
plt.grid(axis='y', alpha=0.75)
plt.xlabel("Clutch Score")
plt.ylabel("Number of Players")
plt.title("Distribution of Clutch Scores")
plt.show()
```



### 1.0.11 Threshold for Clutch Scores

It makes sense to label “clutch” goalscorers as a higher percentile of data. Thus, all players who had a clutch score in the 85th percentile were in the positive class. This approach already highlights the potential shortcomings of classification for this project. Is a player in the 80 to 84th percentile suddenly not “clutch”? Even if we used a multiclass classification approach, how can we distinguish between players who fall near the boundaries?

```
[26]: threshold_elite = merged_clutch_goals['clutch_score'].quantile(0.85)

def label_clutchness(row):
    clutch_score = row['clutch_score']
    if clutch_score >= threshold_elite:
        return 1
    else:
        return 0

merged_clutch_goals['clutch_label'] = merged_clutch_goals.
    ↪ apply(label_clutchness, axis=1)
```

### 1.0.12 Class Imbalance

Due to the right skew distribution of the data, there are very few goalscorers classified as “clutch”.

```
[28]: merged_clutch_goals['clutch_label'].value_counts()
```

```
[28]: clutch_label
0      430
1       76
Name: count, dtype: int64
```

### 1.0.13 Setting up a Classification Model

My initial approach was to select various classification models (e.g. XGBoost, random forest, KNN) and compare them with the Friedman statistical test. I started working on an XGBoost model, but then realized that a classification approach was noideal.

### 1.0.14 Starting with XGBoost

A full glossary of the features can be found on the [Natural Stat Trick website](#).

```
[31]: x_var = ['shots_per_game', 'ixG_per_game', 'iFF_per_game', 'iSCF_per_game',
    ↪ 'iHDCF_per_game',
    ↪ 'assists_per_game', 'iCF_per_game', 'rebounds_created_per_game',
    ↪ 'time_on_ice_per_game',
    ↪ 'off_zone_starts_per_game', 'on_the_fly_starts_per_game']
y_var = 'clutch_label'

X = merged_clutch_goals[x_var]
```

```

y = merged_clutch_goals[y_var]

train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.2,
↳stratify = y)
xgb_model = xgb.XGBClassifier(n_estimators=100, eval_metric='logloss')
xgb_model.fit(train_x, train_y)

```

```

[31]: XGBClassifier(base_score=None, booster=None, callbacks=None,
                    colsample_bylevel=None, colsample_bynode=None,
                    colsample_bytree=None, device=None, early_stopping_rounds=None,
                    enable_categorical=False, eval_metric='logloss',
                    feature_types=None, gamma=None, grow_policy=None,
                    importance_type=None, interaction_constraints=None,
                    learning_rate=None, max_bin=None, max_cat_threshold=None,
                    max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
                    max_leaves=None, min_child_weight=None, missing=nan,
                    monotone_constraints=None, multi_strategy=None, n_estimators=100,
                    n_jobs=None, num_parallel_tree=None, random_state=None, ...)

```

### 1.0.15 Inflated Accuracy

The model's accuracy appears to be quite high (approximately 90%), but this is most likely due to the high class imbalance. The model can predict the majority class most of the time, without effectively learning to identify the minority class.

The model seems to have a high precision and low recall. It is very cautious about predicting the minority class (clutch goalscorers), which results in fewer false positives. So when the model predicts positive, it is mostly correct. However, this means that the model misses many clutch goalscorers and has a low recall.

The F1 score is pulled down by the low recall to highlight the model's issues with rarely predicting the positive class and missing clutch goalscorers.

```

[33]: skf = StratifiedKFold(n_splits=10)

scoring = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score, zero_division=0),
    'recall': make_scorer(recall_score, zero_division=0),
    'f1': make_scorer(f1_score, zero_division=0)
}

scores = cross_validate(xgb_model, X, y, cv = skf, scoring = scoring)

df_scores = pd.DataFrame.from_dict(scores)

df_scores.mean()

```

```
[33]: fit_time      0.043405
      score_time   0.011734
      test_accuracy 0.895451
      test_precision 0.843116
      test_recall   0.621429
      test_f1       0.651007
      dtype: float64
```

### 1.0.16 Learning Curves

The learning curves plot the log loss of the training against the log loss for cross-validation. The very low log loss for training indicates that the model has nearly 100% accuracy in predicting clutch players from the training data. However, the log loss increases to 0.4 on the cross-validation data. Due to the high negative class imbalance, the model can just predict non-clutch most of the time. When it predicts the positive class, it may not be confident enough which shows the model has memorized the patterns in the training data and cannot generalize to new data during cross-validation Note: The high imbalance in the dataset means that stratified cross-validation may not be able to create balanced splits, leading to the error message.

```
[35]: cv = StratifiedKFold(n_splits=10)

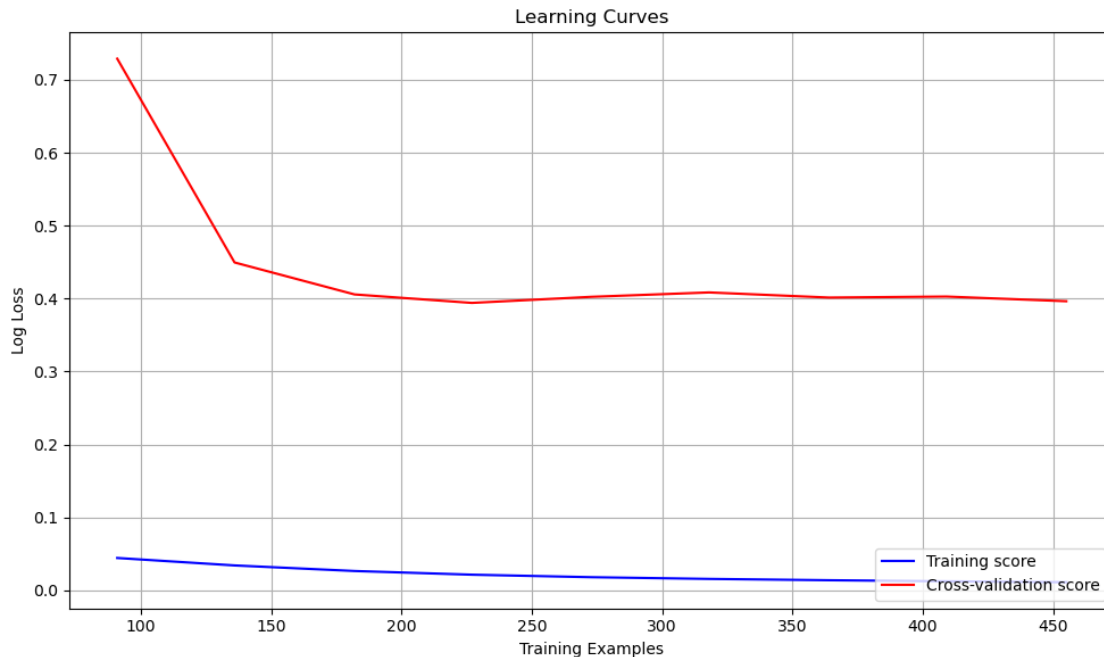
train_sizes = np.linspace(0.1, 1.0, 10)

train_sizes, train_scores, valid_scores = learning_curve(
    xgb_model, X, y,
    cv=cv,
    n_jobs=-1,
    train_sizes=train_sizes,
    scoring='neg_log_loss'
)

train_mean = -np.mean(train_scores, axis=1)
train_std = -np.std(train_scores, axis=1)
valid_mean = -np.mean(valid_scores, axis=1)
valid_std = -np.std(valid_scores, axis=1)

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_mean, label='Training score', color='blue')
plt.plot(train_sizes, valid_mean, label='Cross-validation score', color='red')

plt.title(f'Learning Curves')
plt.xlabel('Training Examples')
plt.ylabel('Log Loss')
plt.grid(True)
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```



### 1.0.17 Hyperparameter tuning

Hyperparameter tuning involves adjusting parameters to improve the model's metrics and reduce overfitting. These parameters are set before training since the model cannot learn them from the data. Below are hyperparameters that are tuned for the XGBoost model better generalization.

```
[37]: from scipy.stats import randint, uniform

param_grid = {
    'max_depth': randint(2, 6),
    'min_child_weight': randint(2, 4),
    'n_estimators': randint(200, 301),
    'learning_rate': uniform(0.03, 0.01),
    'reg_alpha': uniform(0.75, 0.6),
    'reg_lambda': uniform(0.75, 0.6),
    'subsample': uniform(0.7, 0.3),
    'colsample_bytree': uniform(0.7, 0.3)
}
```

### 1.0.18 Random Search

I have repeated random search multiple times on different train and test splits to obtain a good representation of the model's performance. After each train and test split, the model's class weights are adjusted.

### 1.0.19 Results of Hyperparameter Tuning

From the learning curves, it seems that hyperparameter tuning has helped to reduce overfitting.

With regards to the model's performance metrics, it is simply not enough to look at the recall and precision score. We must understand where the model is misclassifying clutch players.

After each randomly selected train test split, I printed out the model's classification results. It appears that the model can correctly classify higher ranked players but struggles with players close to the boundary points (ranks between 45 and 74). The model also incorrectly classifies players with varying performance over the three seasons.

This makes sense because we are essentially assigning an ambiguous label to a clutch player. Is a player on the 84th to 83rd percentile suddenly not clutch? Classification may also have difficulties detecting trends in player performance.

```
[40]: from sklearn.model_selection import RandomizedSearchCV

cv = StratifiedKFold(n_splits=10)

precision_list = []
recall_list = []
f1_list = []

def plot_learning_curves(estimator, X, y, cv, iteration, title):

    train_sizes = np.linspace(0.1, 1.0, 10)

    train_sizes, train_scores, valid_scores = learning_curve(
        estimator, X, y,
        cv=cv,
        n_jobs=-1,
        train_sizes=train_sizes,
        scoring='neg_log_loss'
    )

    train_mean = -np.mean(train_scores, axis=1)
    train_std = -np.std(train_scores, axis=1)
    valid_mean = -np.mean(valid_scores, axis=1)
    valid_std = -np.std(valid_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, label='Training score', color='blue')

    plt.plot(train_sizes, valid_mean, label='Cross-validation score',
    color='red')

    plt.title(f'Learning Curves - Iteration {iteration}\n{title}')
    plt.xlabel('Training Examples')
```

```

plt.ylabel('Log Loss')
plt.ylim(0, 0.5)
plt.grid(True)
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()

for _ in range(5):
    rs = np.random.randint(1, 1000)

    train_x, test_x, train_y, test_y = train_test_split(
        X,
        y,
        test_size=0.2,
        stratify=y,
        random_state = rs
    )

    class_weights = compute_sample_weight(class_weight='balanced', y=train_y)

    xgb_model_adjusted = xgb.XGBClassifier(n_estimators = 100, eval_metric = 'logloss')
    xgb_model_adjusted.fit(train_x, train_y, sample_weight = class_weights)

    random_search = RandomizedSearchCV(xgb_model_adjusted, param_grid, cv=cv, n_iter=20, n_jobs = -1, scoring = 'f1')

    new = random_search.fit(train_x, train_y)

    xgb_best_model = new.best_estimator_

    title = f'Best Parameters: {random_search.best_params_}'
    plot_learning_curves(xgb_best_model, train_x, train_y, cv, _+1, title)

    y_pred = xgb_best_model.predict(test_x)
    y_pred_prob = xgb_best_model.predict_proba(test_x)

    precision = precision_score(test_y, y_pred, zero_division=0)
    recall = recall_score(test_y, y_pred)
    f1 = f1_score(test_y, y_pred)

    print("")
    print("Precision Score: ", precision)
    print("Recall Score: ", recall)
    print("")

```

```

results = pd.DataFrame({
    'Player': merged_clutch_goals.loc[test_y.index, 'Player'],
    'clutch_score_rank': merged_clutch_goals.loc[test_y.index, 'clutch_score_rank'],
    'Actual': test_y,
    'Predicted': y_pred,
})

print("Correct Classifications")
print(results.loc[(results['Actual'] == 1) & (results['Predicted'] == 1)])

print("")

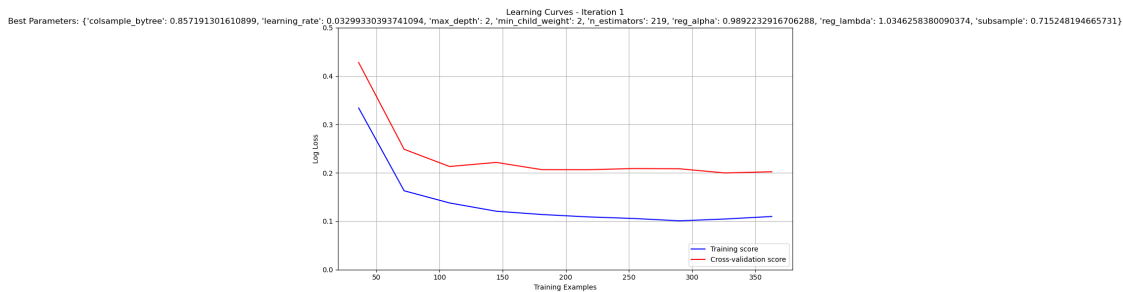
print("Missed Clutch Players")
print(results.loc[(results['Actual'] == 1) & (results['Predicted'] == 0)])

print("")

precision_list.append(precision)
recall_list.append(recall)
f1_list.append(f1)

print("Average Precision:", np.mean(precision_list))
print("Average Recall:", np.mean(recall_list))
print("Average F1 Score:", np.mean(f1_list))

```



Precision Score: 0.7333333333333333

Recall Score: 0.7333333333333333

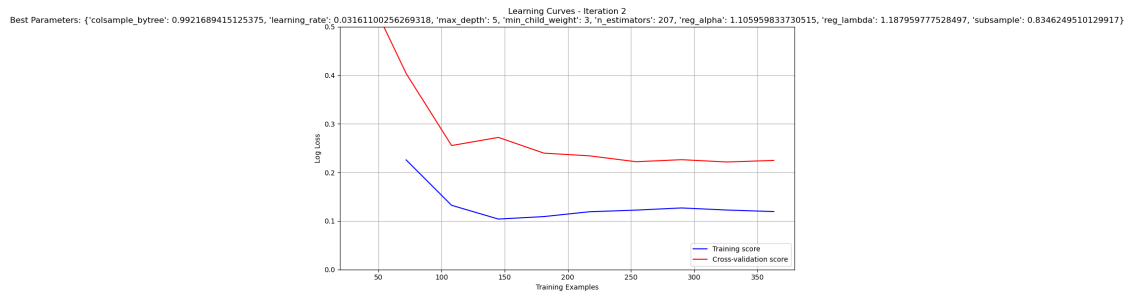
Correct Classifications

	Player	clutch_score_rank	Actual	Predicted
185	Carter Verhaeghe	67.0	1	1
16	Sidney Crosby	28.0	1	1

11	Evgeni Malkin	57.0	1	1
372	Nico Hischier	44.0	1	1
325	Clayton Keller	25.0	1	1
229	Dylan Larkin	14.0	1	1
413	Brady Tkachuk	30.0	1	1
268	Jack Eichel	27.0	1	1
245	Brayden Point	8.0	1	1
267	Connor McDavid	5.0	1	1
324	Patrik Laine	42.0	1	1

#### Missed Clutch Players

	Player	clutch_score_rank	Actual	Predicted
390	Josh Norris	13.0	1	0
228	Jakub Vrana	34.0	1	0
93	Vladimir Tarasenko	76.0	1	0
67	Matt Duchene	55.0	1	0



Precision Score: 0.8461538461538461

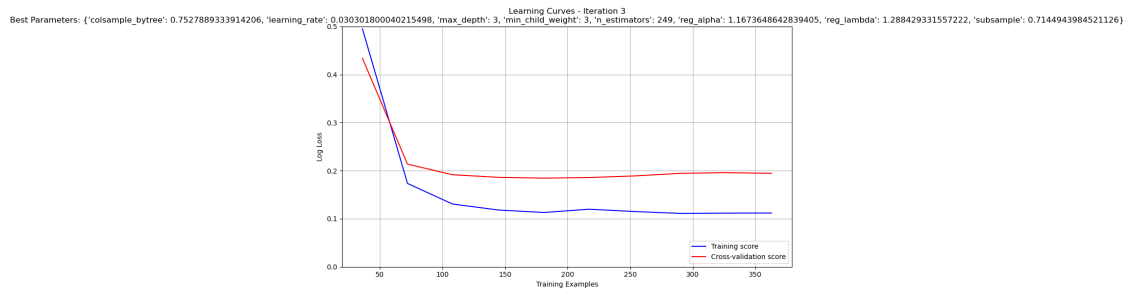
Recall Score: 0.7333333333333333

#### Correct Classifications

	Player	clutch_score_rank	Actual	Predicted
221	Sam Reinhart	19.0	1	1
318	Auston Matthews	1.0	1	1
267	Connor McDavid	5.0	1	1
121	Boone Jenner	39.0	1	1
201	Nathan MacKinnon	18.0	1	1
130	Mika Zibanejad	29.0	1	1
182	Jake Guentzel	50.0	1	1
23	Brad Marchand	58.0	1	1
290	Mitch Marner	37.0	1	1
133	J.T. Miller	53.0	1	1
346	Tage Thompson	9.0	1	1

#### Missed Clutch Players

	Player	clutch_score_rank	Actual	Predicted
381	Nick Suzuki	65.0	1	0
428	Kirill Marchenko	56.0	1	0
185	Carter Verhaeghe	67.0	1	0
389	Martin Necas	69.0	1	0



Precision Score: 0.6363636363636364

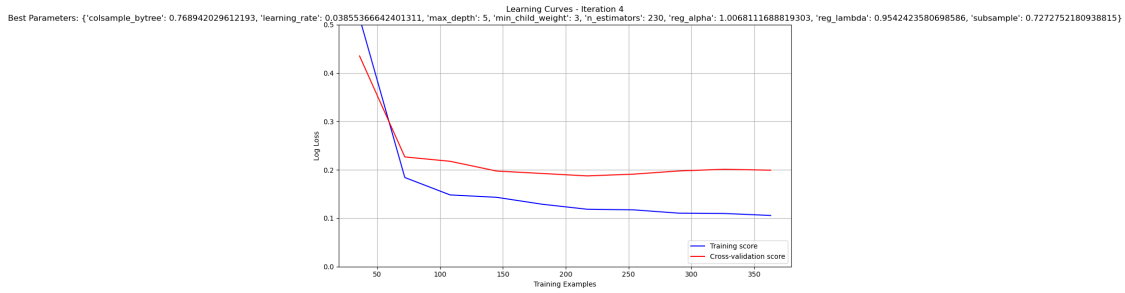
Recall Score: 0.4666666666666667

#### Correct Classifications

	Player	clutch_score_rank	Actual	Predicted
227	Kevin Fiala	62.0	1	1
225	Nikolaj Ehlers	68.0	1	1
268	Jack Eichel	27.0	1	1
236	David Pastrnak	2.0	1	1
185	Carter Verhaeghe	67.0	1	1
133	J.T. Miller	53.0	1	1
413	Brady Tkachuk	30.0	1	1

#### Missed Cltuch Players

	Player	clutch_score_rank	Actual	Predicted
93	Vladimir Tarasenko	76.0	1	0
379	Gabriel Vilardi	46.0	1	0
235	Jared McCann	60.0	1	0
283	Roope Hintz	16.0	1	0
381	Nick Suzuki	65.0	1	0
343	Jesper Bratt	74.0	1	0
390	Josh Norris	13.0	1	0
67	Matt Duchene	55.0	1	0



Precision Score: 0.6666666666666666

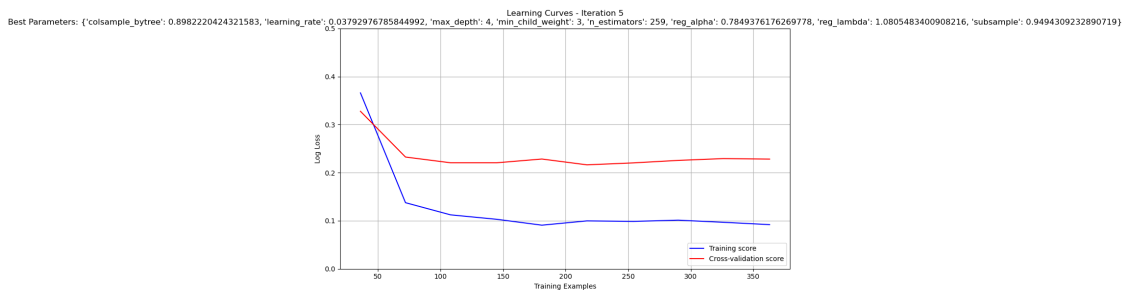
Recall Score: 0.8

#### Correct Classifications

	Player	clutch_score_rank	Actual	Predicted
271	Mikko Rantanen	17.0	1	1
298	Artemi Panarin	24.0	1	1
291	Joel Eriksson Ek	70.0	1	1
225	Nikolaj Ehlers	68.0	1	1
130	Mika Zibanejad	29.0	1	1
339	Jordan Kyrrou	59.0	1	1
304	Kirill Kaprizov	3.0	1	1
449	Cole Caufield	31.0	1	1
236	David Pastrnak	2.0	1	1
68	Evander Kane	49.0	1	1
72	Chris Kreider	32.0	1	1
11	Evgeni Malkin	57.0	1	1

#### Missed Cltuch Players

	Player	clutch_score_rank	Actual	Predicted
307	Troy Terry	36.0	1	0
510	Andrei Kuzmenko	44.0	1	0
131	Mark Scheifele	23.0	1	0



Precision Score: 0.7058823529411765

Recall Score: 0.8

#### Correct Classifications

	Player	clutch_score_rank	Actual	Predicted
290	Mitch Marner	37.0	1	1
274	Sebastian Aho	22.0	1	1
43	Patrick Kane	47.0	1	1
68	Evander Kane	49.0	1	1
229	Dylan Larkin	14.0	1	1
413	Brady Tkachuk	30.0	1	1
325	Clayton Keller	25.0	1	1
227	Kevin Fiala	62.0	1	1
511	Connor Bedard	33.0	1	1
150	Tomas Hertl	73.0	1	1
222	Leon Draisaitl	4.0	1	1
291	Joel Eriksson Ek	70.0	1	1

#### Missed Clutch Players

	Player	clutch_score_rank	Actual	Predicted
185	Carter Verhaeghe	67.0	1	0
109	Mark Stone	63.0	1	0
235	Jared McCann	60.0	1	0

Average Precision: 0.7176799670917318

Average Recall: 0.7066666666666667

Average F1 Score: 0.7069563769563769

### 1.0.20 Switching to Regression

Although the classification model does show advantages in correctly classifying some player, I believe that regression is more suitable:

1. Unlike Classification, regression can be used to predict the player's clutch score (a continuous label), rather than assigning them to classes that may not clearly define a "clutch player". This makes the model easier to interpret and leads to more accurate predictions.
2. Regression can account for the trends in player performance and provide better predictions.

### 1.0.21 Features

The same features from classification are used. These features show a strong positive correlation with clutch score, which indicates that a linear regression model is suitable

```
[43]: x_var = ['shots_per_game', 'ixG_per_game', 'iFF_per_game', 'iSCF_per_game',  
            ↪ 'iHDCF_per_game',  
            'assists_per_game', 'iCF_per_game', 'rebounds_created_per_game',  
            ↪ 'time_on_ice_per_game',  
            'off_zone_starts_per_game']
```

```

X= merged_clutch_goals[x_var]
y_var = 'clutch_score'
y = merged_clutch_goals[y_var]

correlation = X.corrwith(y)
print(correlation)

```

```

shots_per_game          0.864417
ixG_per_game            0.854483
iFF_per_game            0.869267
iSCF_per_game           0.876447
iHDCF_per_game          0.694483
assists_per_game        0.745530
iCF_per_game            0.863786
rebounds_created_per_game 0.781373
time_on_ice_per_game     0.771436
off_zone_starts_per_game 0.744034
dtype: float64

```

### 1.0.22 Scatter Plots

The scatter plots further show the strong positive correlation of the features with clutch score.

```

[45]: plt.figure(figsize=(15, 12))

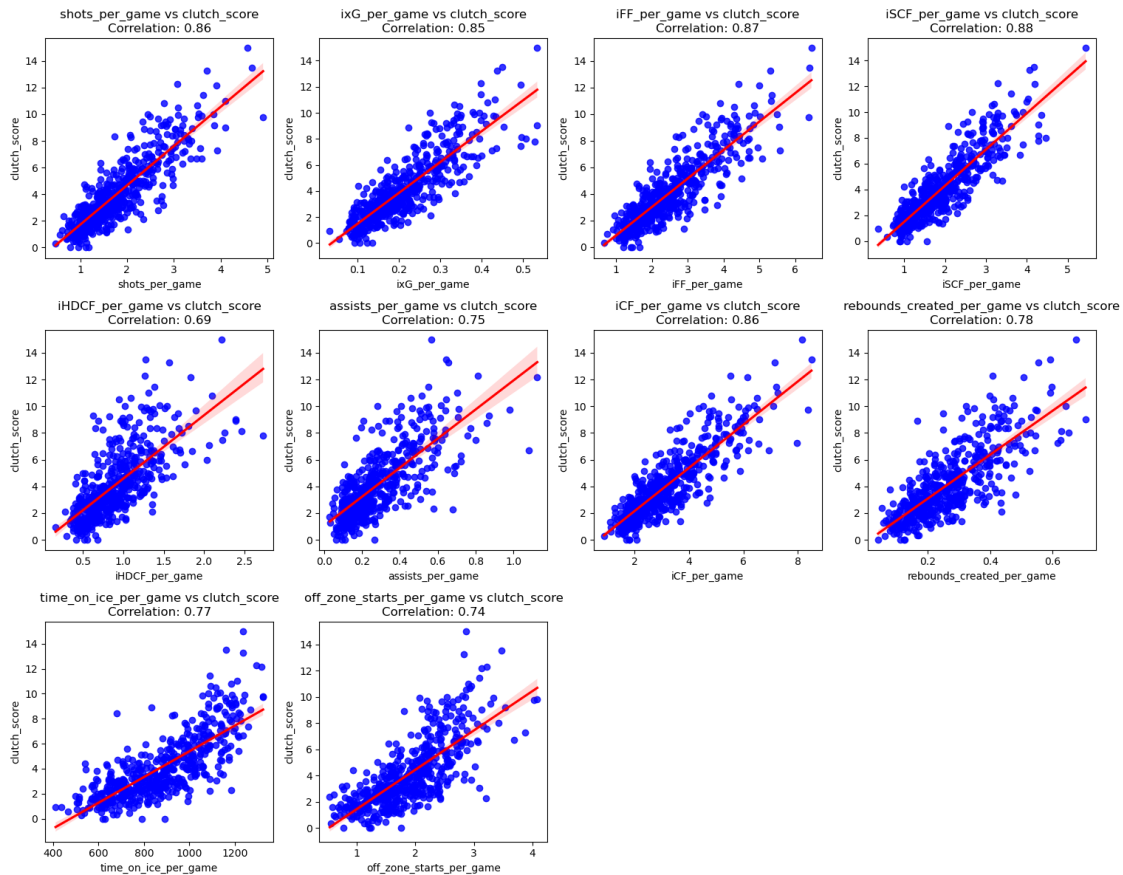
for i, var in enumerate(x_var):
    plt.subplot(3, 4, i+1)

    sns.regplot(data=merged_clutch_goals, x=var, y=y, scatter_kws={'color': 'blue'}, line_kws={'color': 'red'})

    plt.title(f'{var} vs {y_var}\nCorrelation: {correlation[var]:.2f}',
    ↪fontsize=12)
    plt.xlabel(var)
    plt.ylabel(y_var)

plt.tight_layout()
plt.show()

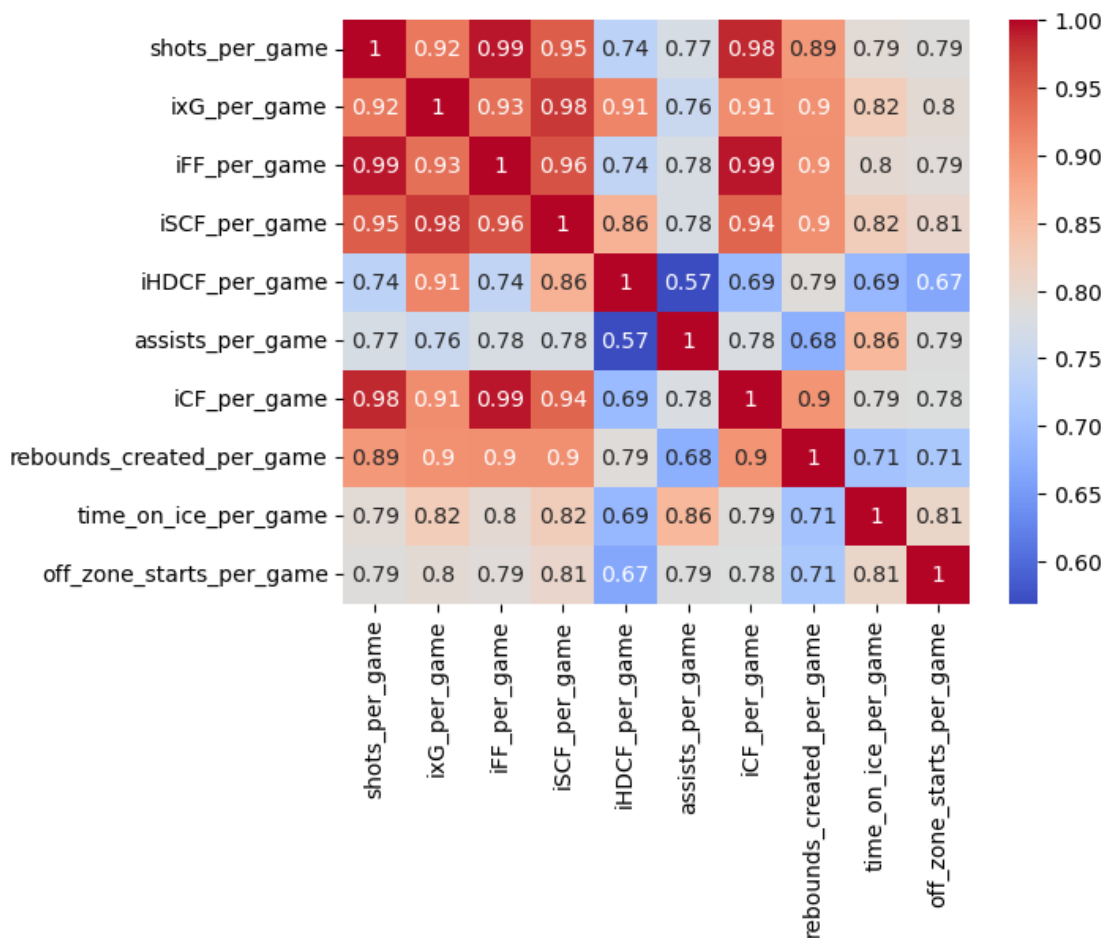
```



### 1.0.23 Multicollinearity

As the heatmap shows, there is high multicollinearity among features, which would lead to instability in coefficients and make it difficult to interpret the impact of features on the clutch score. Therefore, a small subset of features were kept (scoring chances, assists, time on ice, rebounds created, offensive zone starts).

```
[47]: sns.heatmap(X.corr(), annot=True, cmap='coolwarm')
plt.show()
```



### 1.0.24 Ridge Regression

Ridge regression is used to ensure there is less overfitting. The model shows good performance because it has a low MSE of approximately 1 and  $R^2$  of approximately 80%. In future sections, the outliers are evaluated to determine the model's limitations which are not obvious with the MSE and  $R^2$ .

```
[49]: x_var = ['iSCF_per_game', 'assists_per_game', 'rebounds_created_per_game',
            ↪ 'time_on_ice_per_game', 'off_zone_starts_per_game']

X_adjusted = merged_clutch_goals[x_var]
y_var = 'clutch_score'
y = merged_clutch_goals[y_var]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_adjusted)
```

```

train_x, test_x, train_y, test_y = train_test_split(X_scaled, y, test_size=0.2,
    random_state=42)

alphas = np.logspace(-3, 3, 20)

ridge_cv = RidgeCV(alphas=alphas, cv=5)
ridge_cv.fit(train_x, train_y)

y_pred = ridge_cv.predict(test_x)

mse = mean_squared_error(test_y, y_pred)
rmse = np.sqrt(mse)
median_error = median_absolute_error(test_y, y_pred)
r2 = r2_score(test_y, y_pred)

print("MSE: ", mse)
print("RMSE: ", rmse)
print("Median Error: ", median_error)
print("R2: ", r2)
print("Adjusted R2: ", 1 - (1 - r2) * (len(train_y) - 1) / (len(train_y) -
    train_x.shape[1] - 1))

```

```

MSE:  1.816778370521922
RMSE:  1.3478792121410295
Median Error:  0.7598699884636728
R2:  0.7824620529478941
Adjusted R2:  0.7797291641658324

```

### 1.0.25 Learning Curves

The learning curves do not show significant overfitting. After approximately 250 samples, both training and validation curves converge to an MSE of less than 2. Thus, Ridge Regression is the correct choice for generalizing the training data.

```

[51]: train_sizes = np.linspace(0.1, 1.0, 10)

train_sizes, train_scores, validation_scores = learning_curve(
    ridge_cv,
    X_scaled,
    y, train_sizes=train_sizes, cv=10,
    scoring='neg_mean_squared_error')

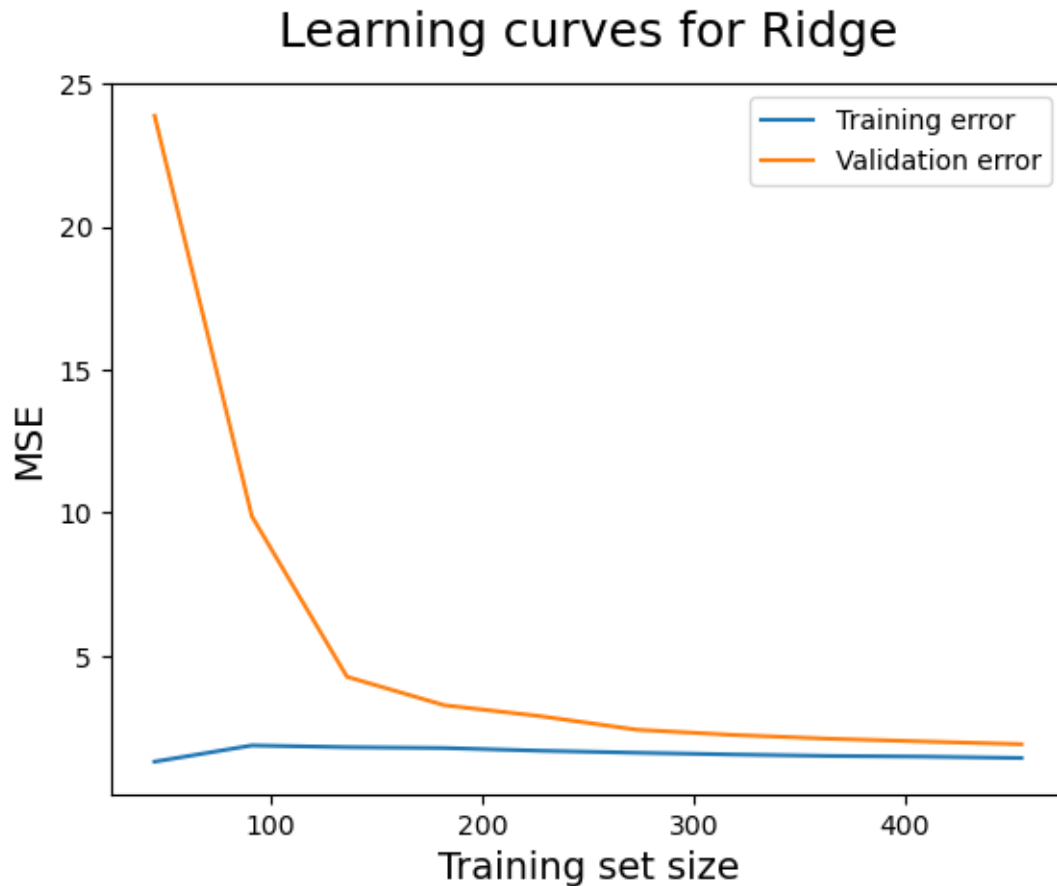
train_scores_mean = -train_scores.mean(axis=1)
validation_scores_mean = -validation_scores.mean(axis=1)

plt.plot(train_sizes, train_scores_mean, label='Training error')
plt.plot(train_sizes, validation_scores_mean, label='Validation error')
plt.ylabel('MSE', fontsize=14)

```

```
plt.xlabel('Training set size', fontsize = 14)
plt.title('Learning curves for Ridge', fontsize = 18, y = 1.03)
plt.legend()
```

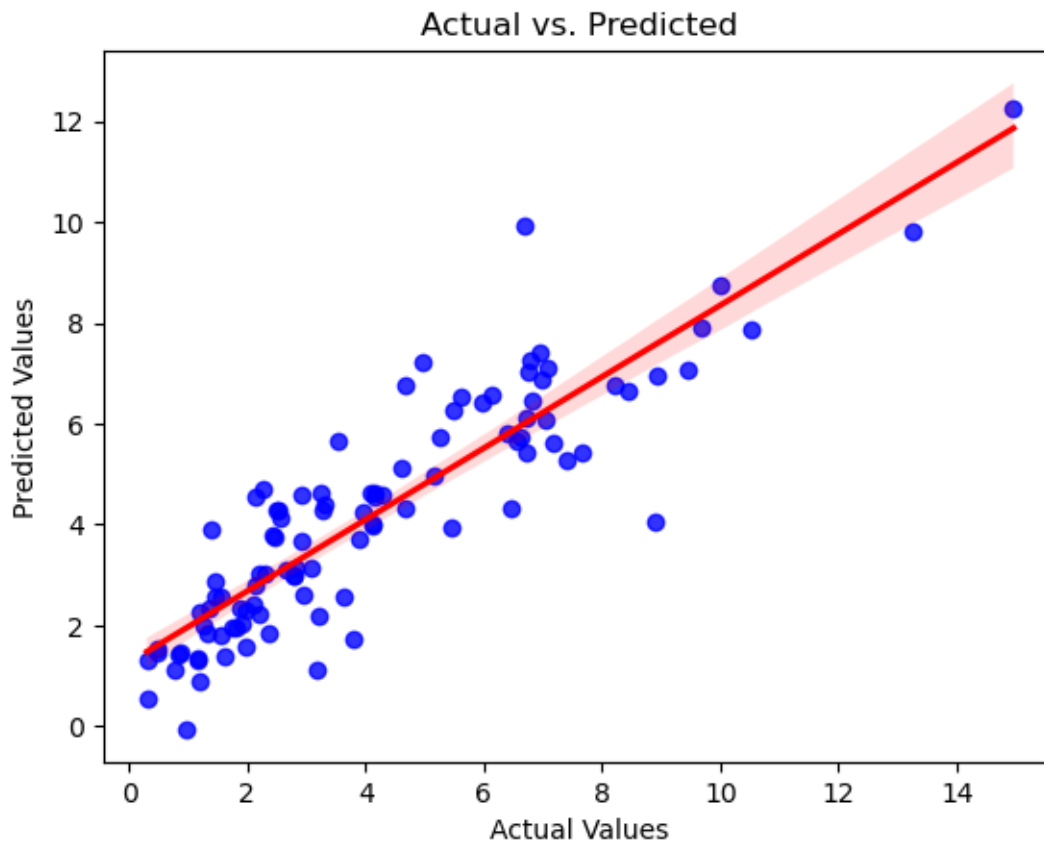
[51]: <matplotlib.legend.Legend at 0x274483cf950>



### 1.0.26 Scatter Plot and Line of Best Fit

Since most points fall near the line of best fit, the model is generally accurate in predicting values. However, there are a few outliers which need to be corrected.

```
[53]: sns.regplot(data=merged_clutch_goals, x=test_y, y=y_pred, scatter_kws={'color': 'blue'}, line_kws={'color': 'red'})
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted')
plt.show()
```

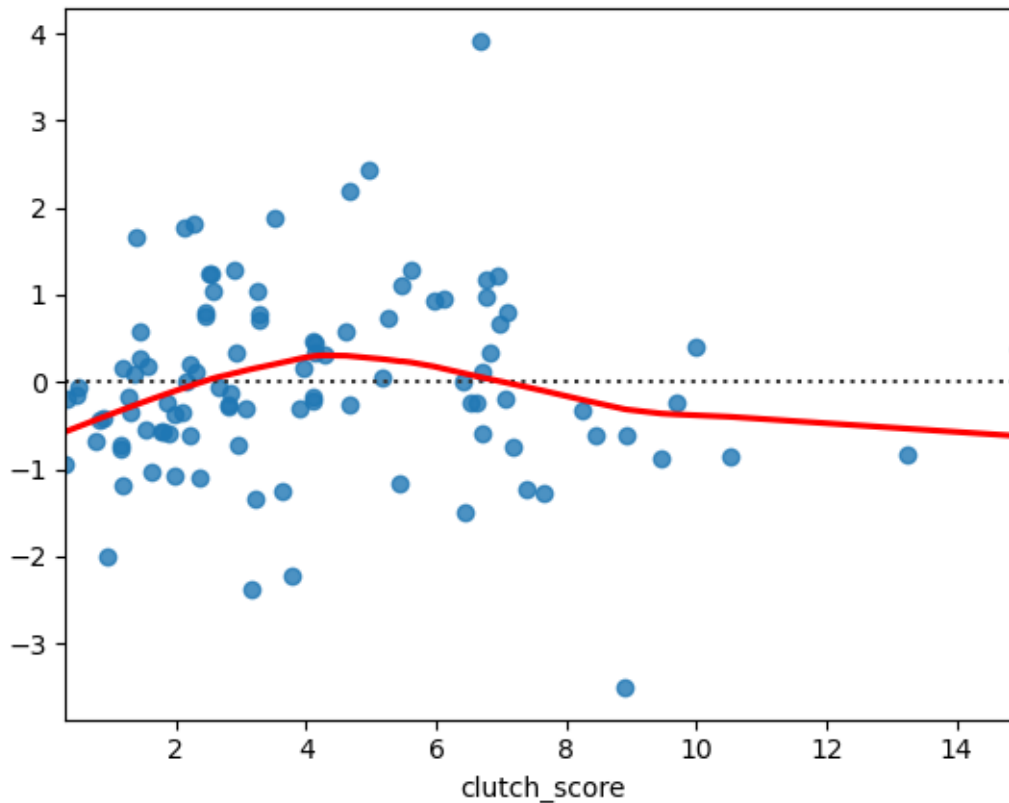


### 1.0.27 Residual Plot

The residual plot shows more errors in predicting the clutch score are between 1 and -1. However, there are a few points outside of this range, which may be considered as outliers.

```
[55]: sns.residplot(data=merged_clutch_goals, x=test_y, y=y_pred, lowess=True,
↳ line_kws=dict(color="r"))
```

```
[55]: <Axes: xlabel='clutch_score'>
```



### 1.0.28 Cook's Distance

Cook's distance enables us to evaluate influential points in the model. Influential points are data points that significantly change the fit of the model if removed.

As shown below, the model tends to underestimate the performance of several elite players (e.g., McDavid and Matthews) in clutch situations. These players' statistics may have created an artificial "ceiling" that limits the model's ability to accurately predict their scoring ability in close and tied situations.

Conversely, the model overestimates the performance of other elite players (e.g., Matthew Tkachuk), who do not perform as well in clutch scoring situations as their general statistics suggest.

```
[57]: X_with_intercept = sm.add_constant(X_scaled)

ols_model = sm.OLS(y, X_with_intercept).fit()

influence = ols_model.get_influence()
cooks_d, _ = influence.cooks_distance

threshold = 4 / len(X_adjusted)
outliers = np.where(cooks_d > threshold)[0]
```

```

results = pd.DataFrame({
    'Player': merged_clutch_goals.loc[y.index, 'Player'],
    'Actual': y,
    'Predicted': ols_model.fittedvalues,
    'Cook\'s Distance': cooks_d
})

outliers_df = results.loc[results["Cook's Distance"] > threshold]

print("There are", outliers_df.shape[0], "influential points.")
print("Outliers based on Cook's Distance:")
print(outliers_df)

plt.figure(figsize=(10, 6))
plt.stem(results.index, cooks_d, markerfmt='b.', label="Cook's Distance")
plt.axhline(y=threshold, color='r', linestyle='--', label=f"Threshold:␣
↳{threshold:.4f}")
plt.xlabel("Player ID")
plt.ylabel("Cook's Distance")
plt.title("Cook's Distance for Each Data Point")
plt.legend()
plt.show()

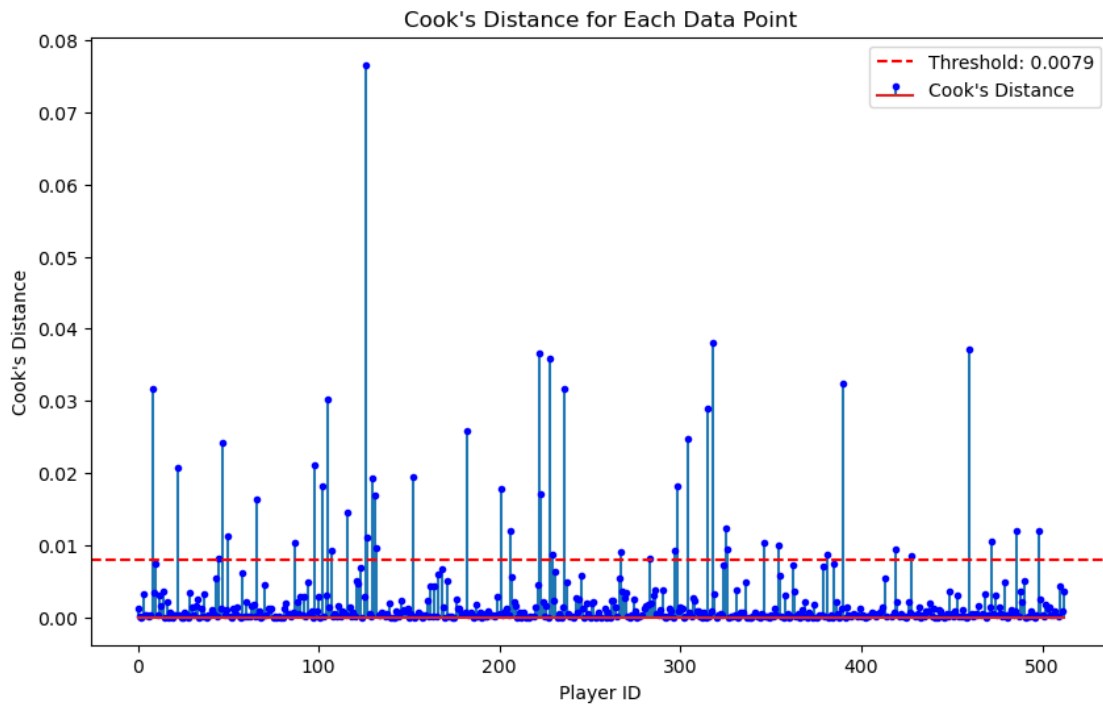
```

There are 45 influential points.

Outliers based on Cook's Distance:

	Player	Actual	Predicted	Cook's Distance
318	Auston Matthews	14.96	13.027892	0.037989
236	David Pastrnak	13.50	10.221958	0.031744
304	Kirill Kaprizov	13.25	10.023435	0.024740
222	Leon Draisaitl	12.26	8.619711	0.036664
267	Connor McDavid	12.16	11.258861	0.009111
152	Filip Forsberg	11.44	8.672784	0.019481
346	Tage Thompson	10.64	7.553515	0.010428
50	Steven Stamkos	10.52	7.794900	0.011200
390	Josh Norris	10.00	6.041798	0.032389
229	Dylan Larkin	9.93	7.956490	0.008622
283	Roope Hintz	9.83	7.294436	0.008227
201	Nathan MacKinnon	9.75	11.465788	0.017908
206	Bo Horvat	9.61	7.510754	0.011934
131	Mark Scheifele	9.46	7.309917	0.016853
298	Artemi Panarin	9.27	7.609395	0.018181
325	Clayton Keller	9.23	6.297872	0.012383
130	Mika Zibanejad	9.10	6.138103	0.019318
228	Jakub Vrana	8.89	4.337478	0.035828
460	Pavel Dorofeyev	8.43	4.644457	0.037233
66	John Tavares	8.14	10.137546	0.016284
182	Jake Guentzel	8.01	10.784396	0.025763

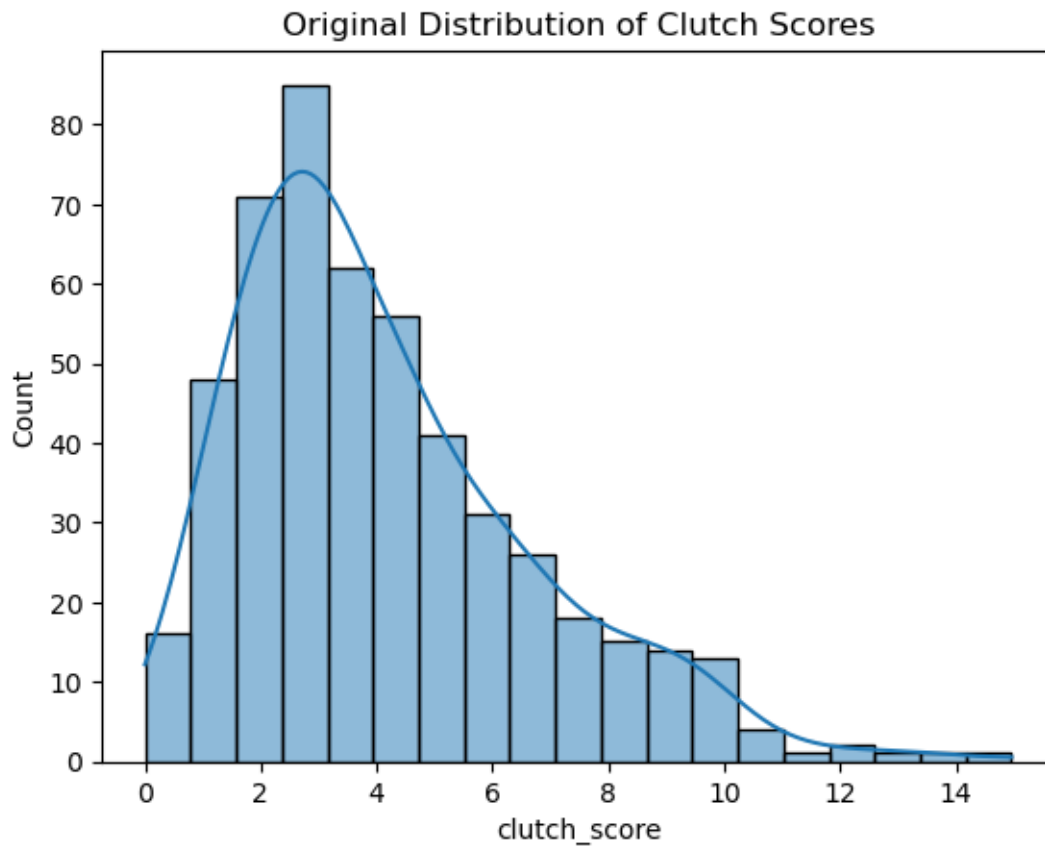
98	Zach Hyman	7.79	10.083087	0.021132
428	Kirill Marchenko	7.66	5.781781	0.008579
315	Matthew Tkachuk	7.47	9.759691	0.028978
381	Nick Suzuki	7.34	5.455565	0.008762
8	Patrice Bergeron	6.69	9.150043	0.031713
126	Nikita Kucherov	6.67	9.311853	0.076586
472	Connor Zary	6.35	3.465775	0.010569
419	Andrei Svechnikov	5.15	7.453933	0.009407
127	Ryan Nugent-Hopkins	4.96	7.126555	0.011048
116	Vincent Trocheck	4.73	7.873103	0.014490
486	Alexander Holtz	4.68	2.087303	0.011992
223	Sam Bennett	4.66	7.707330	0.017109
498	Walker Duehr	4.41	1.719857	0.011976
132	Sean Couturier	3.54	5.285874	0.009581
107	Brendan Gallagher	3.38	5.288197	0.009257
45	Mikael Backlund	3.17	5.791964	0.008065
87	Evgeny Kuznetsov	3.07	4.971118	0.010269
297	Evan Rodrigues	2.79	5.411284	0.009193
102	Mikael Granlund	2.30	4.809059	0.018102
326	Jesse Puljujarvi	2.28	4.291107	0.009470
47	Jakub Voracek	2.28	4.097403	0.024271
22	Patric Hornqvist	2.13	4.513678	0.020735
354	Michael Eyssimont	1.36	3.626404	0.009987
105	Joonas Donskoi	0.00	3.286230	0.030226

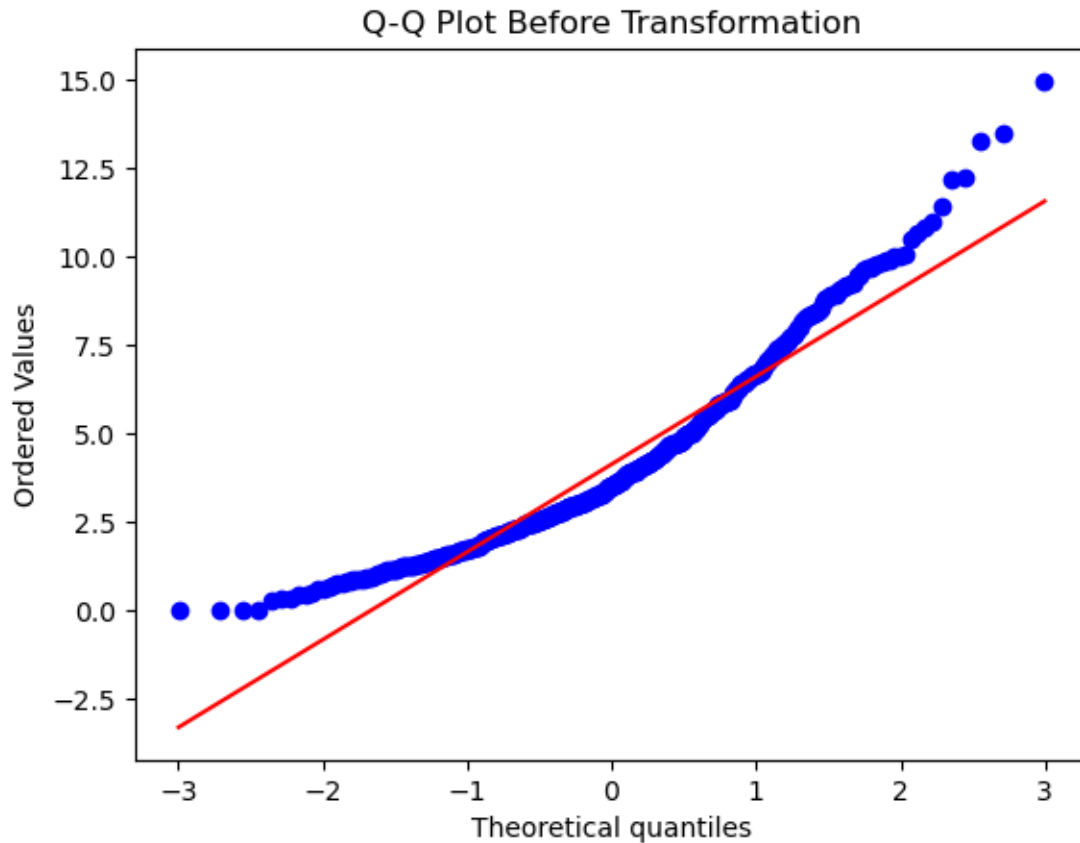


### 1.0.29 Evaluating the Distribution of the Data

The histogram and QQ plot show that the data has a right skew distribution, which may explain why the model has difficulties in predicting the clutch score of elite players on the right side of the tail.

```
[59]: sns.histplot(y, kde=True)
plt.title("Original Distribution of Clutch Scores")
plt.show()
stats.probplot(y, dist="norm", plot=plt)
plt.title("Q-Q Plot Before Transformation")
plt.show()
```





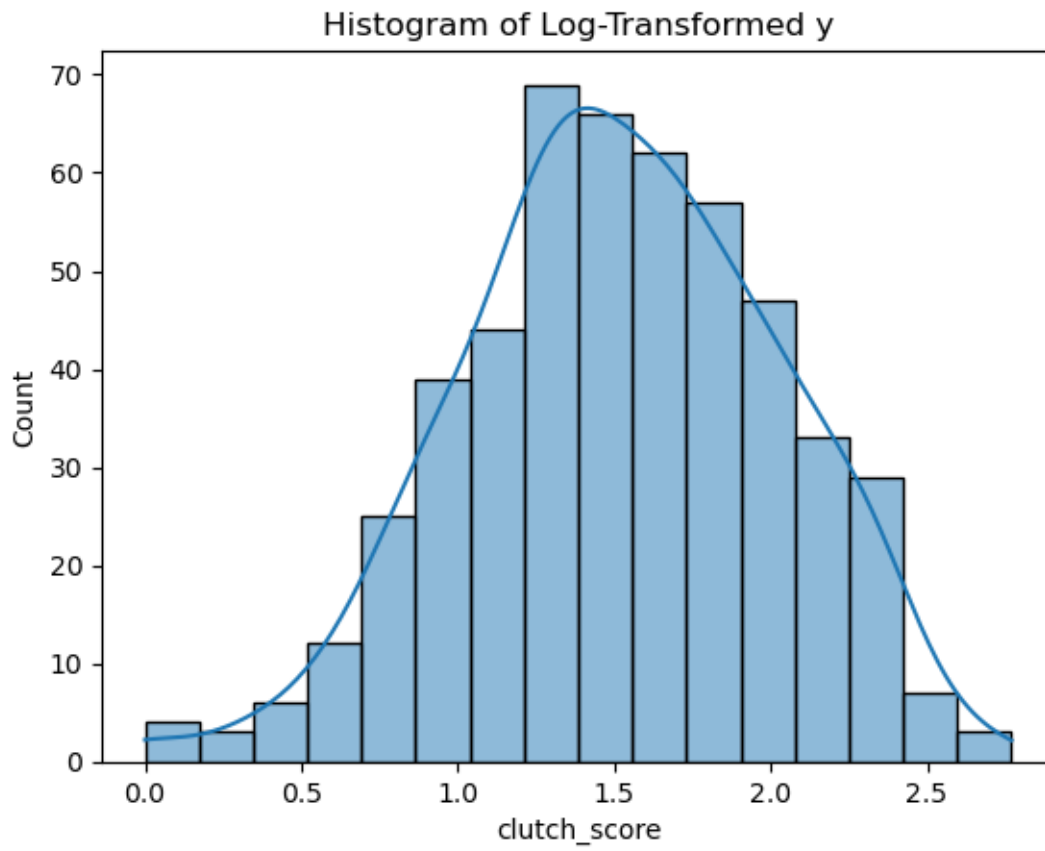
### 1.0.30 Transforming the Data to a Normal Distribution with Log

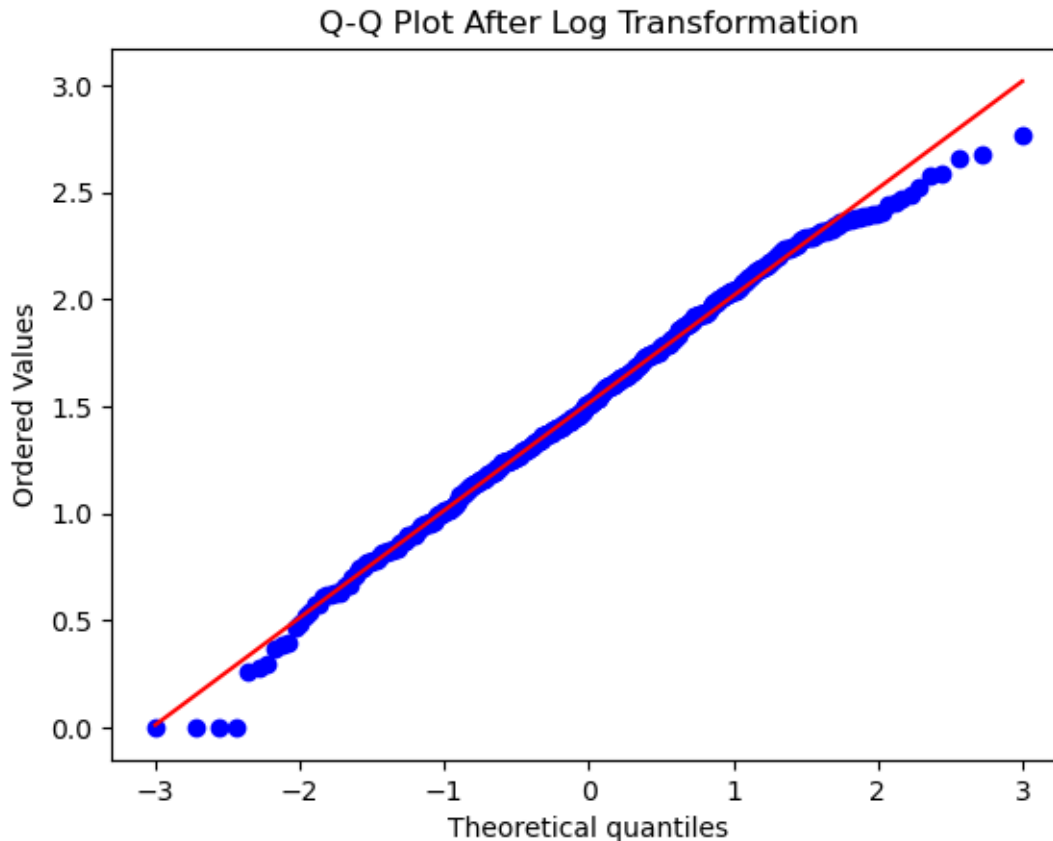
As shown below, a log transformation is used to reduce the skew of the data and create a normal distribution. This ensures the predictions are not affected by the influential points we identified in Cook's distance.

```
[61]: y_log = np.log(y + 1)

sns.histplot(y_log, kde=True)
plt.title("Histogram of Log-Transformed y")
plt.show()

stats.probplot(y_log, dist="norm", plot=plt)
plt.title("Q-Q Plot After Log Transformation")
plt.show()
```





### 1.0.31 Calculating Cook's Distance

After we apply the log transformation and calculate Cook's distance, we can see that the elite players are no longer influential points. However, there are some players which the model still struggles with. The model undervalues some players (e.g. Vrana, Laine) who may perform better in close and tied situations than their metrics suggest. On the other hand, some players are overvalued and may have better metrics that may not fully reflect their clutch performance (e.g. Matthew Tkachuk, Nikita Kucherov). While influential points are often viewed negatively, they can provide valuable insights. These points could help NHL coaching staff and management identify players who perform well in high-pressure situations, even if they aren't considered elite based on traditional metrics.

Finally, some below-average players become influential because the log transformation tends to amplify the difference between smaller actual and predicted values.

```
[63]: epsilon = np.abs(X_scaled.min()) + 1
X_shifted = X_scaled + epsilon
y_log = np.log(y + 1)
X_log = np.log(X_shifted)

train_x, test_x, train_y, test_y = train_test_split(
    X_log,
```

```

    y_log,
    test_size=0.2,
    random_state=200
)

alphas = np.logspace(-3, 3, 20)
ridge_cv_log = RidgeCV(alphas=alphas, cv=5)
ridge_cv_log.fit(train_x, train_y)
y_pred = ridge_cv_log.predict(test_x)

```

```

[64]: X_with_intercept = sm.add_constant(X_log)

ols_model = sm.OLS(y_log, X_with_intercept).fit()

influence = ols_model.get_influence()
cooks_d, _ = influence.cooks_distance

threshold = 4 / len(X_with_intercept)

results = pd.DataFrame({
    'Player': merged_clutch_goals.loc[y.index, 'Player'],
    'Actual': y_log,
    'Predicted': ols_model.fittedvalues,
    'Cook\'s Distance': cooks_d
})

outliers_df = results.loc[results["Cook's Distance"] > threshold]

print("There are", outliers_df.shape[0], "influential points.")
print("Outliers based on Cook's Distance:")
print(outliers_df)

plt.figure(figsize=(10, 6))
plt.stem(results.index, cooks_d, markerfmt='b.', label="Cook's Distance")
plt.axhline(y=threshold, color='r', linestyle='--', label=f"Threshold:␣
↪{threshold:.4f}")
plt.xlabel("Player ID")
plt.ylabel("Cook's Distance")
plt.title("Cook's Distance for Each Data Point")
plt.legend()
plt.show()

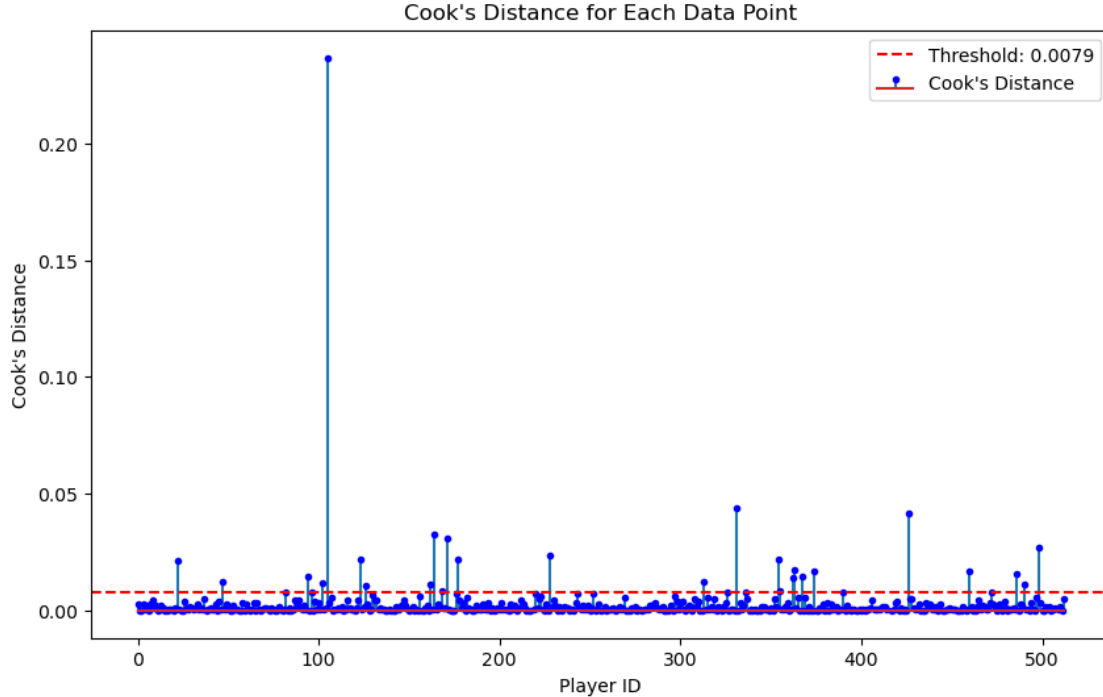
```

There are 26 influential points.

Outliers based on Cook's Distance:

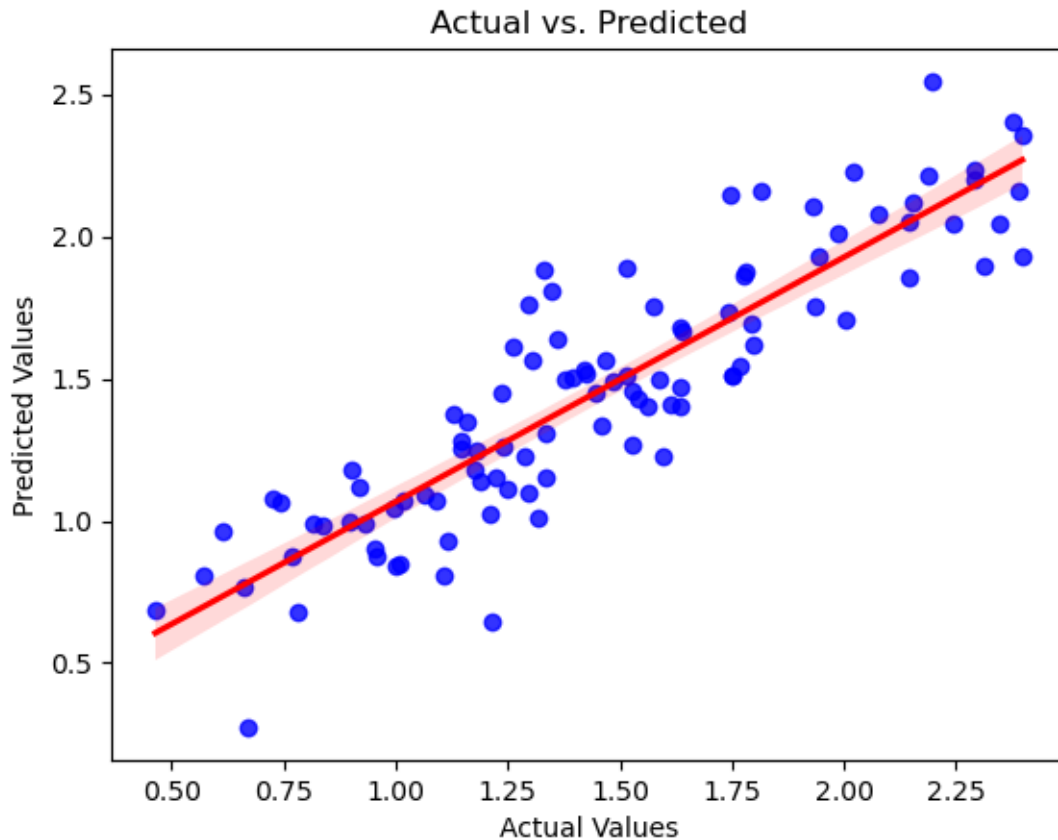
	Player	Actual	Predicted	Cook's Distance
228	Jakub Vrana	2.291524	1.563531	0.023625
460	Pavel Dorofeyev	2.243896	1.703203	0.017004
126	Nikita Kucherov	2.037317	2.345150	0.010375

355	Michael Carcone	1.890095	1.277452	0.008042
486	Alexander Holtz	1.736951	1.177567	0.015785
498	Walker Duehr	1.688249	1.006267	0.026980
362	Vinni Lettieri	1.654411	0.947276	0.014116
490	Will Cuyllle	1.597365	1.227084	0.010899
123	Brett Ritchie	1.566530	0.988282	0.021923
94	Austin Watson	1.427916	0.811915	0.014430
363	Mason Shaw	1.269761	0.808278	0.017283
164	Dominic Toninato	1.214913	0.717981	0.032529
102	Mikael Granlund	1.193922	1.676579	0.011728
47	Jakub Voracek	1.187843	1.497233	0.012313
22	Patric Hornqvist	1.141033	1.650758	0.021421
168	Connor Brown	0.875469	1.543848	0.008278
354	Michael Eyssimont	0.858662	1.474342	0.021929
162	Teddy Blueger	0.774727	1.262328	0.011380
171	Kurtis MacDermid	0.667829	0.267013	0.030769
367	Jake Leschyshyn	0.371564	0.863724	0.014534
331	Beck Malenstyn	0.292670	0.976549	0.043992
313	Kevin Rooney	0.277632	0.888848	0.012184
105	Joonas Donskoi	0.000000	1.294005	0.236797
374	Jonas Rondbjerg	0.000000	0.947130	0.016984
426	Bo Groulx	0.000000	0.886523	0.041519
177	Saku Maenalanen	0.000000	0.801358	0.021906



### 1.0.32 Final Scatter Plot from Training

```
[66]: sns.regplot(data=merged_clutch_goals, x=test_y, y=y_pred, scatter_kws={'color': 'blue'}, line_kws={'color': 'red'})
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted')
plt.show()
```



### 1.0.33 Making Predictions on Current Season Data

We save “ridge\_cv\_log” for reproducible results. We can then use it to make predictions on the current statistics of players (from 2024-2025 season to the current 2025-2026 season).

```
[68]: joblib.dump(ridge_cv_log, 'ridge_cv_model.pkl')
ridge_cv_log_loaded = joblib.load('ridge_cv_model.pkl')

joblib.dump(scaler, 'scaler.pkl')
joblib.dump(epsilon, 'epsilon.pkl')
```

```
[68]: ['epsilon.pkl']
```

```
[69]: all_seasons = []

for season in range(2024, 2026):
    summary_url = f"https://api.nhle.com/stats/rest/en/skater/summary?
    ↪limit=-1&cayenneExp=seasonId={season}-{season+1}%20and%20gameTypeId=2"

    try:
        summary_resp = requests.get(summary_url)
        summary_resp.raise_for_status()
        summary_json = summary_resp.json()

        if summary_json['data']:
            df_summary = pd.DataFrame(summary_json['data'])
            all_seasons.append(df_summary)
            df_summary['season'] = f"{season}-{season + 1}"
            print(f"Successfully fetched data for season {season}-{season+1}")
        else:
            print(f"No data returned for season {season}-{season + 1}")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching data for season {season}-{season + 1}: {e}")

if all_seasons:
    nhl_api_df = pd.concat(all_seasons, ignore_index=True)
    nhl_api_df = nhl_api_df.groupby('playerId').agg({
        'playerId': 'first',
        'skaterFullName': 'first',
        'positionCode': 'first',
        'gamesPlayed': 'sum',
        'goals': 'sum',
        'assists': 'sum',
        'otGoals': 'sum',
        'timeOnIcePerGame': 'mean',
        'teamAbbrevs': 'last'
    }).reset_index(drop = True)

print(nhl_api_df)
```

Successfully fetched data for season 2024-2025

Successfully fetched data for season 2025-2026

	playerId	skaterFullName	positionCode	gamesPlayed	goals	assists	\
0	8470600	Ryan Suter	D	82	2	13	
1	8470613	Brent Burns	D	119	11	37	
2	8470621	Corey Perry	R	112	26	20	
3	8471214	Alex Ovechkin	L	103	59	47	
4	8471215	Evgeni Malkin	C	94	24	55	

...	...	...	...	...	...	...
1004	8485483	Karsen Dorwart	L	5	0	0
1005	8485493	David Tomasek	R	22	3	2
1006	8485511	Quinn Hutson	R	5	1	0
1007	8485512	Tim Washe	C	2	0	0
1008	8485702	Max Shabanov	R	25	3	7

	otGoals	timeOnIcePerGame	teamAbbrevs
0	0	1168.28040	STL
1	0	1213.52205	COL
2	0	764.64335	LAK
3	1	1064.67810	WSH
4	1	1058.75845	PIT

...	...	...	...
1004	0	658.80000	PHI
1005	0	645.50000	EDM
1006	0	629.75000	EDM
1007	0	464.00000	ANA
1008	0	839.16000	NYI

[1009 rows x 9 columns]

```
[70]: nhl_api_df = nhl_api_df.loc[(nhl_api_df['positionCode'] != 'D') &
    ↪(nhl_api_df['gamesPlayed'] >= 40)]
nhl_api_df = nhl_api_df.reset_index(drop = True)

rename_columns = {
    'otGoals': 'ot_goals',
    'skaterFullName': 'Player',
    'timeOnIcePerGame': 'time_on_ice_per_game'
}

nhl_api_df.rename(columns = rename_columns, inplace = True)
```

```
[71]: start_season = "20242025"
end_season = "20252026"
goals_up_one_url = f"https://www.naturalstatattrick.com/playerteams.php?
    ↪fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=u1&stdoi=std&rate=n
goals_down_one_url = f"https://www.naturalstatattrick.com/playerteams.php?
    ↪fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=d1&stdoi=std&rate=n
tied_url = f"https://www.naturalstatattrick.com/playerteams.php?
    ↪fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=tied&stdoi=std&rate=
total_url = f"https://www.naturalstatattrick.com/playerteams.php?
    ↪fromseason={start_season}&thruseason={end_season}&stype=2&sit=all&score=all&stdoi=std&rate=
on_ice_url = f"https://www.naturalstatattrick.com/playerteams.php?
    ↪fromseason={start_season}&thruseason={end_season}&stype=2&sit=5v5&score=all&stdoi=oi&rate=n
```

```
[72]: urls = {
    "goals_up_one": (goals_up_one_url, 'goals_up_by_one'),
    "goals_down_one": (goals_down_one_url, 'goals_down_by_one'),
    "tied": (tied_url, 'goals_when_tied'),
    "total": (total_url, 'total_goals'),
    "on_ice": (on_ice_url, '')
}

dataframes = {}

for name, (url, new_column_name) in urls.items():
    df = pd.read_html(url, header=0, index_col=0, na_values=["-"])[0]
    df.rename(columns={'Goals': new_column_name}, inplace=True)
    dataframes[name] = df

goals_up_one_df = dataframes["goals_up_one"]
goals_down_one_df = dataframes["goals_down_one"]
goals_tied_df = dataframes["tied"]
total_df = dataframes["total"]
on_ice_df = dataframes["on_ice"]
on_ice_df.columns = on_ice_df.columns.str.replace('\xa0', ' ')

[73]: goals_up_one_df = goals_up_one_df[['Player', 'GP', 'goals_up_by_one']]
goals_down_one_df = goals_down_one_df[['Player', 'goals_down_by_one']]
goals_tied_df = goals_tied_df[['Player', 'goals_when_tied']]
total_df = total_df[['Player', 'total_goals', 'Shots', 'ixG', 'iFF', 'iSCF',
    ↪ 'iHDCF', 'Rebounds Created', 'iCF']]
on_ice_df = on_ice_df[['Player', 'Off. Zone Starts', 'On The Fly Starts']]

dfs_natural_stat = [goals_up_one_df, goals_down_one_df, goals_tied_df,
    ↪ total_df, on_ice_df]

merged_natural_stat = ft.reduce(lambda left, right: pd.merge(left, right,
    ↪ on='Player'), dfs_natural_stat)
merged_natural_stat = merged_natural_stat.loc[merged_natural_stat['GP'] >= 40]

rename_columns = {
    'Shots': 'shots',
    'Rebounds Created': 'rebounds_created',
    'Off. Zone Starts': 'off_zone_starts',
    'On The Fly Starts': 'on_the_fly_starts'
}
merged_natural_stat.rename(columns = rename_columns, inplace=True)
```

[74]:

```
natural_stat_names = ["Pat Maroon", "Alex Kerfoot", "Nicholas Paul", "Zach
↳Sanford", "Alex Wennberg", "Mitchell Marner", "Zach Aston-Reese", "Max
↳Comtois", "Alexei Toropchenko", "Cameron Atkinson", "Alexander Nylander",
↳"Jacob Lucchini", "Zack Bolduc", "Frederic Gaudreau"]
nhl_names = ["Patrick Maroon", "Alexander Kerfoot", "Nick Paul", "Zachary
↳Sanford", "Alexander Wennberg", "Mitch Marner", "Zachary Aston-Reese",
↳"Maxime Comtois", "Alexey Toropchenko", "Cam Atkinson", "Alex Nylander",
↳"Jake Lucchini", "Zachary Bolduc", "Freddy Gaudreau" ]
merged_natural_stat = merged_natural_stat.replace(natural_stat_names, nhl_names)
```

```
[75]: merged_clutch_goals_prediction = nhl_api_df.merge(merged_natural_stat, on =
↳'Player', how = 'left')
merged_clutch_goals_prediction.drop(columns = 'GP', axis = 1, inplace = True)
merged_clutch_goals_prediction = merged_clutch_goals_prediction.dropna()
```

```
[76]: columns = ['ot_goals', 'assists', 'goals_up_by_one', 'goals_down_by_one',
↳'goals_when_tied', 'shots', 'ixG', 'iFF', 'iSCF', 'iHDCF', 'iCF',
↳'rebounds_created', 'off_zone_starts', 'on_the_fly_starts']
for column in columns:
    per_game_string = f"{column}_per_game"
    merged_clutch_goals_prediction[per_game_string] =
↳merged_clutch_goals_prediction[column] /
↳merged_clutch_goals_prediction['gamesPlayed']
```

```
[77]: merged_clutch_goals_prediction['clutch_score'] = (
    0.45 * merged_clutch_goals_prediction['goals_down_by_one_per_game'] +
    0.35 * merged_clutch_goals_prediction['goals_when_tied_per_game'] +
    0.2 * merged_clutch_goals_prediction['ot_goals_per_game']
)
```

```
[78]: merged_clutch_goals_prediction['clutch_score'] *= 100
merged_clutch_goals_prediction['clutch_score_rank'] =
↳merged_clutch_goals_prediction['clutch_score'].rank(ascending = False,
↳method = 'min')
merged_clutch_goals_prediction['clutch_score'] =
↳merged_clutch_goals_prediction['clutch_score'].apply(lambda x: round(x, 2))
merged_clutch_goals_prediction.sort_values('clutch_score_rank', inplace = True)
merged_clutch_goals_prediction[['Player', 'clutch_score', 'clutch_score_rank']].
↳head(20)
```

```
[78]:
```

	Player	clutch_score	clutch_score_rank
149	Leon Draisaitl	15.18	1.0
212	Kirill Kaprizov	11.81	2.0
226	Alex DeBrincat	11.74	3.0
391	Dylan Guenther	11.70	4.0
253	Morgan Geekie	11.34	5.0
324	Cole Caufield	11.18	6.0

240	Tage Thompson	10.88	7.0
27	John Tavares	10.80	8.0
136	Bo Horvat	10.79	9.0
289	Brady Tkachuk	10.56	10.0
268	Josh Norris	10.45	11.0
3	Sidney Crosby	10.39	12.0
94	Tom Wilson	10.34	13.0
148	Sam Reinhart	10.22	14.0
78	Mark Scheifele	10.17	15.0
151	William Nylander	9.96	16.0
265	Jason Robertson	9.88	17.0
161	Adrian Kempe	9.87	18.0
224	Auston Matthews	9.75	19.0
73	Nikita Kucherov	9.73	20.0

```
[79]: x_var = ['iSCF_per_game', 'assists_per_game', 'rebounds_created_per_game',
            'time_on_ice_per_game', 'off_zone_starts_per_game']
X_adjusted = merged_clutch_goals_prediction[x_var]
y_var = 'clutch_score'
y = merged_clutch_goals_prediction[y_var]

scaler = joblib.load('scaler.pkl')
epsilon = joblib.load('epsilon.pkl')

X_scaled = scaler.transform(X_adjusted)
X_scaled = np.nan_to_num(X_scaled, nan=0)

X_shifted = X_scaled + epsilon
X_log = np.log(X_shifted)

y_log = np.log(y + 1)
y_pred = ridge_cv_log_loaded.predict(X_log)
```

### 1.0.34 Evaluating the Model after Testing

The  $R^2$  indicates the model explains approximately 65% of variance in clutch performance, which is strong given the inherent randomness in clutch situations.

```
[81]: r2 = r2_score(y_log, y_pred)
rmse = np.sqrt(mean_squared_error(y_log, y_pred))
mae = mean_absolute_error(y_log, y_pred)

print(f"Test Set Performance:")
print(f"R2: {r2:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAE: {mae:.4f}")
```

Test Set Performance:

$R^2$ : 0.6486

RMSE: 0.3280

MAE: 0.2423

```
[82]: y_pred = ridge_cv_log_loaded.predict(X_log)
merged_clutch_goals_prediction['predicted_clutch_score'] = y_pred

merged_clutch_goals_prediction['log'] = np.
    ↳log(merged_clutch_goals_prediction['clutch_score'] + 1)
merged_clutch_goals_prediction['log_adjusted'] = np.
    ↳log(merged_clutch_goals_prediction['clutch_score'] + 1) * 10
merged_clutch_goals_prediction['log_adjusted'] =
    ↳merged_clutch_goals_prediction['log_adjusted'].apply(lambda x: round(x, 2))
merged_clutch_goals_prediction['predicted_clutch_score_adjusted'] = y_pred * 10
merged_clutch_goals_prediction = merged_clutch_goals_prediction.
    ↳sort_values(by='predicted_clutch_score_adjusted', ascending = False)
merged_clutch_goals_prediction['predicted_clutch_score_adjusted'] =
    ↳merged_clutch_goals_prediction['predicted_clutch_score_adjusted'].
    ↳apply(lambda x: round(x, 2))
```

### 1.0.35 Prediction Intervals

95% prediction intervals were generated for each player. If actual clutch scores fall outside the intervals, this indicates that clutch performance is significantly different from expectations. The intervals are generated using a bootstrap procedure with resampled residual noise, which ensures that the intervals reflect randomness in clutch performance.

```
[84]: n_boot = 1000
alpha = ridge_cv_log_loaded.alpha_

boot_preds = np.zeros((n_boot, len(X_log)))

for i in range(n_boot):
    idx = np.random.choice(len(X_log), size=len(X_log), replace=True)

    X_res = X_log[idx]
    y_res = y_log.iloc[idx]

    ridge = Ridge(alpha=alpha)
    ridge.fit(X_res, y_res)

    preds = ridge.predict(X_log)

    residuals = y_log - ridge_cv_log_loaded.predict(X_log)
    noise = np.random.choice(residuals, size=len(X_log), replace=True)

    boot_preds[i] = preds + noise
```

```

lower_log = np.percentile(boot_preds, 2.5, axis=0)
upper_log = np.percentile(boot_preds, 97.5, axis=0)

merged_clutch_goals_prediction['lower_bound_log'] = (lower_log * 10).round(2)
merged_clutch_goals_prediction['upper_bound_log'] = (upper_log * 10).round(2)

merged_clutch_goals_prediction['Significantly_Clutch'] = np.where(
    (merged_clutch_goals_prediction['log_adjusted'] >=
merged_clutch_goals_prediction['lower_bound_log']) &
    (merged_clutch_goals_prediction['log_adjusted'] <=
merged_clutch_goals_prediction['upper_bound_log']),
    'Inside Range',
    'Outside Range'
)

```

### 1.0.36 Shap Values

SHAP values were calculated to explain which features most influenced each player's prediction. This is useful for the dashboard since users can understand how clutch scores are predicted.

```

[86]: explainer = shap.LinearExplainer(ridge_cv_log_loaded, X_log)
      shap_values = explainer(X_log)

      shap_df = pd.DataFrame(
          shap_values.values,
          columns=X_adjusted.columns,
          index=X_adjusted.index
      )

      for col in shap_df.columns:
          merged_clutch_goals_prediction[f'shap_{col}'] = shap_df[col]

```

### 1.0.37 Cook's Distance Observations

The model shows the same patterns as before - it undervalues and overvalues some players. Clutch scores of low performing players are also amplified by the log transformation. These players are excluded in the final dashboard by only including players with 20+ goals.

```

[88]: X_with_intercept = sm.add_constant(X_log)

      ols_model = sm.OLS(y_log, X_with_intercept).fit()

      influence = ols_model.get_influence()
      cooks_d, _ = influence.cooks_distance

      threshold = 4 / len(X_adjusted)

```

```

merged_clutch_goals_prediction = merged_clutch_goals_prediction.
    ↪reset_index(drop=True)

results = pd.DataFrame({
    'Player': merged_clutch_goals_prediction['Player'].values,
    'Actual': merged_clutch_goals_prediction['log'].values,
    'Predicted': merged_clutch_goals_prediction['predicted_clutch_score'].
    ↪values,
    "Cook's Distance": cooks_d
})

outliers_df = results.loc[results["Cook's Distance"] > threshold]

print("There are", outliers_df.shape[0], "influential points.")
print("Outliers based on Cook's Distance:")
print(outliers_df)

plt.figure(figsize=(10, 6))
plt.stem(results.index, cooks_d, markerfmt='b.', label="Cook's Distance")
plt.axhline(y=threshold, color='r', linestyle='--', label=f"Threshold:␣
    ↪{threshold:.4f}")
plt.xlabel("Player ID")
plt.ylabel("Cook's Distance")
plt.title("Cook's Distance for Each Data Point")
plt.legend()
plt.show()

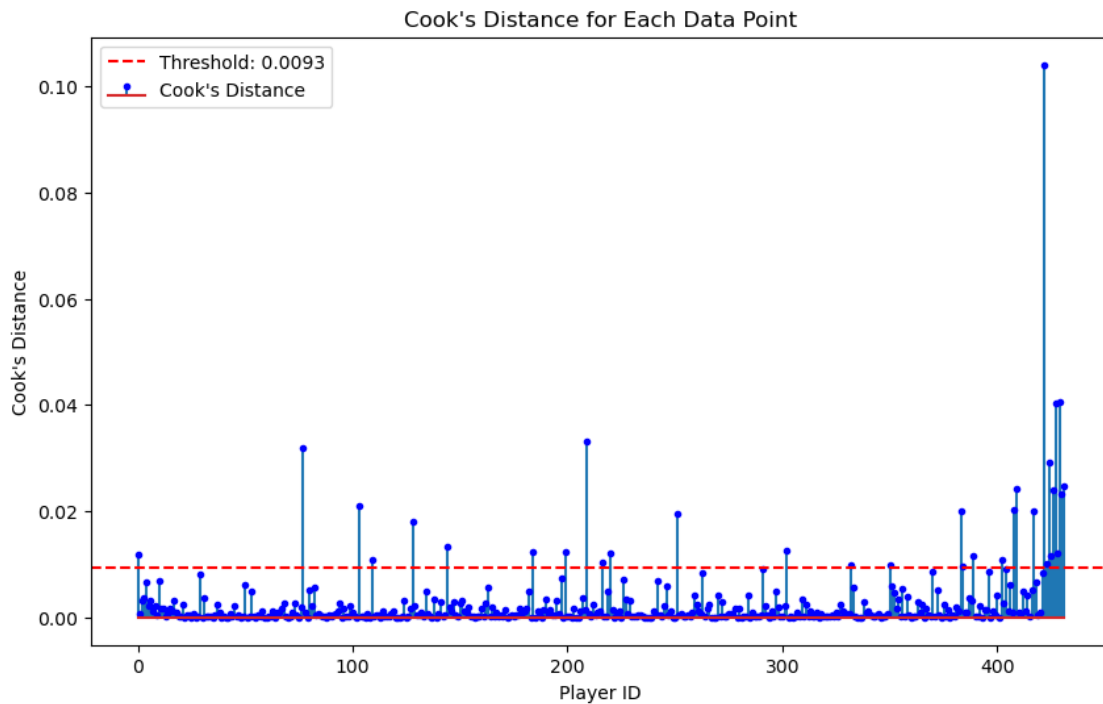
```

There are 32 influential points.

Outliers based on Cook's Distance:

	Player	Actual	Predicted	Cook's Distance
0	Nathan MacKinnon	2.354228	2.566055	0.011729
77	William Eklund	1.835776	1.979267	0.031944
103	Conor Garland	1.821318	1.909293	0.020966
109	Ryan O'Reilly	2.033398	1.902727	0.010720
128	Pierre-Luc Dubois	1.593309	1.827881	0.018107
144	Warren Foegele	1.938742	1.774610	0.013290
184	William Karlsson	1.453953	1.628294	0.012205
199	Jordan Martinook	1.081805	1.579310	0.012269
209	Kiefer Sherwood	2.127041	1.538187	0.033029
216	Nino Niederreiter	1.479329	1.522605	0.010313
220	Christian Dvorak	1.208960	1.509226	0.012142
251	Matt Savoie	0.593327	1.415087	0.019618
302	Noah Laba	1.286474	1.265439	0.012616
332	Colton Dach	1.275363	1.154192	0.009883
350	Alexey Toropchenko	0.518794	1.104594	0.009783
383	Cole Schwindt	0.512824	0.952120	0.020023
384	Radek Faksa	0.722706	0.950664	0.009571

389	Mattias Janmark	0.364643	0.935897	0.011571
402	Adam Edstrom	1.202972	0.863806	0.010919
408	David Kampf	0.867100	0.763551	0.020347
409	Mikael Pyyhtia	0.000000	0.763083	0.024323
417	Cole Reinhardt	0.717840	0.698380	0.020083
422	John Beecher	0.717840	0.645389	0.104125
423	Luke Glendening	0.530628	0.638556	0.010129
424	Rasmus Kupari	0.565314	0.633731	0.029095
425	Devin Shore	0.000000	0.628570	0.011636
426	Ben Jones	0.000000	0.620678	0.023955
427	Kevin Rooney	0.963174	0.619530	0.040448
428	Curtis Lazar	0.512824	0.576785	0.012104
429	Zack Ostapchuk	0.398776	0.547043	0.040494
430	Samuel Helenius	0.845868	0.444271	0.023312
431	Ryan Reaves	0.418710	0.404914	0.024656

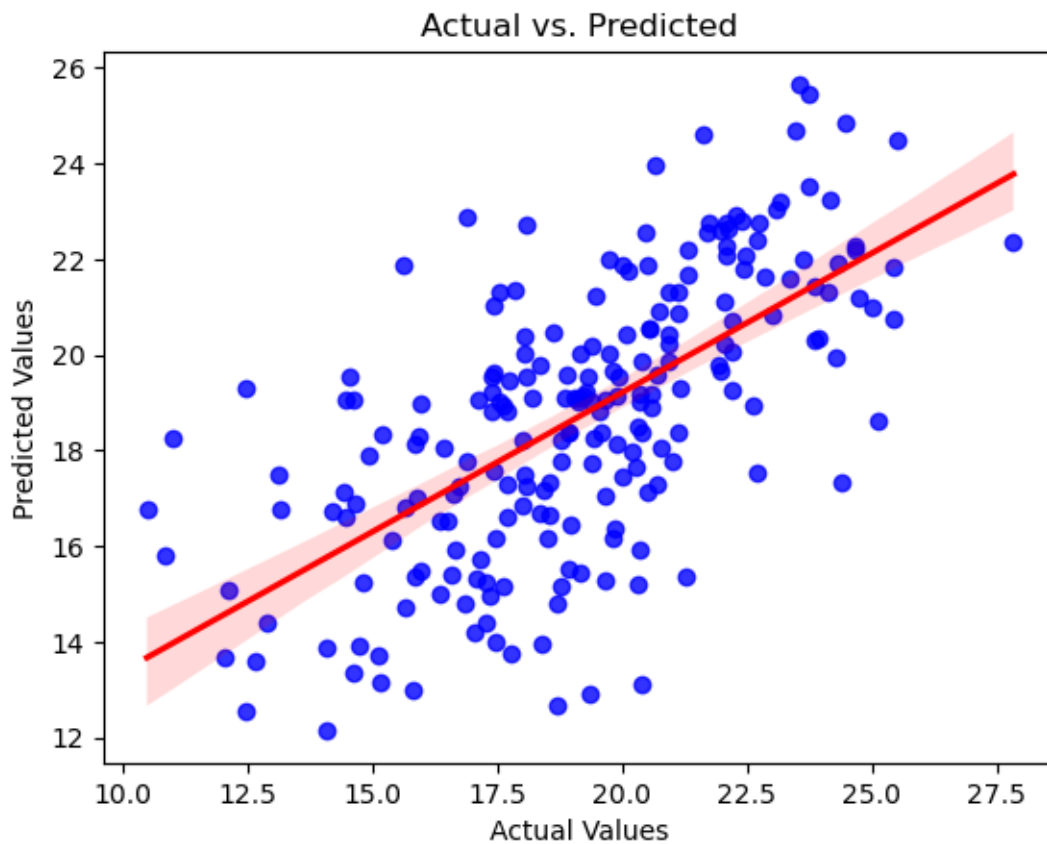


### 1.0.38 Final Scatter Plot after Testing

The Actual vs. Predicted shows a well-fitted model for clutch performance. There is a strong linear relationship and homoscedasticity. Some points may deviate from the line of best fit, but this is to be expected due to players naturally overperforming/underperforming their clutch scores.

[90]:

```
merged_clutch_goals_prediction = merged_clutch_goals_prediction.
    ↳loc[merged_clutch_goals_prediction['total_goals'] >= 20]
sns.regplot(data=merged_clutch_goals_prediction,
    ↳x=merged_clutch_goals_prediction['log_adjusted'],
    ↳y=merged_clutch_goals_prediction['predicted_clutch_score_adjusted'],
    ↳scatter_kws={'color': 'blue'}, line_kws={'color': 'red'})
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted')
plt.show()
```



### 1.0.39 Conclusion

Through this project, I hope that NHL fans can identify forwards who perform well in close game situations and use the regression model to determine if they are underperforming/overperforming expectations. The SHAP analysis should make the model less of a “black box” and enable users to gain more insight into playing styles that influence the predictions. For those more statistically inclined, the prediction intervals can show players who are truly “clutch”.

While there were still some influential points in the final model, these points may be useful in

determining overvalued and undervalued players. One potential limitation in the clutch score is that it includes goals from all periods. It would be useful to have only third period goals given these are the most clutch goals, but I was not able to filter by period on Natural Stat Trick.