In [1]:
```
import pandas as pd
ball = pd.read_csv('Statcast_2021.csv')
ball
```

Out[1]:

	Unnamed: 0	pitch_type	game_date	release_speed	release_pos_x	release_pos_z	player
0	2926	FF	2021-10-03	92.3	1.40	6.80	Sm
1	3048	SL	2021-10-03	80.6	1.60	6.64	Sm
2	3294	CU	2021-10-03	75.5	1.46	6.88	Sm
3	3447	CU	2021-10-03	75.0	1.53	6.83	Sm
4	3545	FF	2021-10-03	91.2	1.49	6.66	Sm
...	...	...	...	...	...	...	
709846	3470	FF	2021-04-01	95.9	-1.92	6.15	M
709847	3684	FF	2021-04-01	95.4	-1.69	6.17	M
709848	3810	FF	2021-04-01	96.1	-1.76	6.27	M
709849	3937	FF	2021-04-01	95.6	-2.04	6.03	M
709850	4026	FF	2021-04-01	95.1	-1.95	6.11	M

709851 rows × 93 columns

In [2]:
```
shohei = ball[ball['player_name']=='Ohtani, Shohei']
shohei
```

Out[2]:

	Unnamed: 0	pitch_type	game_date	release_speed	release_pos_x	release_pos_z	player
29965	0	FF	2021-09-26	99.2	-2.15	5.88	
29966	1	FS	2021-09-26	91.8	-2.10	5.82	
29967	2	SL	2021-09-26	85.5	-2.32	5.74	
29968	3	FF	2021-09-26	97.4	-1.98	5.86	

29969	5	FS	2021-09-26	90.8	-2.00	5.89
...	...	...	...	...	...	...
696532	3106	SL	2021-04-04	83.3	-2.17	5.91
696533	3133	FF	2021-04-04	98.9	-1.88	6.00
696534	3304	FF	2021-04-04	96.4	-1.90	6.03
696535	3381	FF	2021-04-04	96.9	-1.94	6.00
696536	3429	FF	2021-04-04	98.2	-1.92	5.96

2027 rows × 93 columns

In [3]:
```
shohei = shohei[['pitch_type', 'release_speed',
                 'description', 'launch_speed', 'launch_angle',
                'release_spin_rate', 'release_extension']]
shohei
```

Out[3]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rat
29965	FF	99.2	foul_tip	NaN	NaN	2095
29966	FS	91.8	foul	65.7	-48.0	1293
29967	SL	85.5	swinging_strike	NaN	NaN	2358
29968	FF	97.4	foul	75.8	42.0	2008
29969	FS	90.8	ball	NaN	NaN	1347
...	...	...	...	...	...	
696532	SL	83.3	hit_into_play	72.4	1.0	2595
696533	FF	98.9	foul	70.3	-31.0	2377
696534	FF	96.4	swinging_strike	NaN	NaN	2411
696535	FF	96.9	ball	NaN	NaN	2358
696536	FF	98.2	ball	NaN	NaN	2457

2027 rows × 7 columns

In [4]:
```
if ball is not None:
    # Replace NaN in 'launch_speed' with 0
```

```
    shohei['launch_speed'].fillna(0, inplace=True)
    shohei['launch_angle'].fillna(0, inplace=True)
    shohei['release_spin_rate'].fillna(0, inplace=True)
else:
    shohei = "Data loading error: " + error_message

shohei
```

```
/var/folders/4v/_s6ktch56r3dgggt7wmn18dm0000gn/T/ipykernel_16172/1892
539085.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  shohei['launch_speed'].fillna(0, inplace=True)
/var/folders/4v/_s6ktch56r3dgggt7wmn18dm0000gn/T/ipykernel_16172/1892
539085.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  shohei['launch_angle'].fillna(0, inplace=True)
/var/folders/4v/_s6ktch56r3dgggt7wmn18dm0000gn/T/ipykernel_16172/1892
539085.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  shohei['release_spin_rate'].fillna(0, inplace=True)
```

Out[4]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rat
29965	FF	99.2	foul_tip	0.0	0.0	2095
29966	FS	91.8	foul	65.7	-48.0	1293
29967	SL	85.5	swinging_strike	0.0	0.0	2358
29968	FF	97.4	foul	75.8	42.0	2008
29969	FS	90.8	ball	0.0	0.0	1347
...	...	...	...	...	...	
696532	SL	83.3	hit_into_play	72.4	1.0	2595
696533	FF	98.9	foul	70.3	-31.0	2377

696534	FF	96.4	swinging_strike	0.0	0.0	2411
696535	FF	96.9	ball	0.0	0.0	2358
696536	FF	98.2	ball	0.0	0.0	2457

2027 rows × 7 columns

In [5]:
```
na_count = shohei.isna().sum()
na_count
```

Out[5]:
```
pitch_type           0
release_speed        0
description          0
launch_speed         0
launch_angle         0
release_spin_rate    0
release_extension    9
dtype: int64
```

In [6]:
```
shohei = shohei.dropna()
shohei
```

Out[6]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rat
29965	FF	99.2	foul_tip	0.0	0.0	2095
29966	FS	91.8	foul	65.7	-48.0	1293
29967	SL	85.5	swinging_strike	0.0	0.0	2358
29968	FF	97.4	foul	75.8	42.0	2008
29969	FS	90.8	ball	0.0	0.0	1347
...	...	...	...	...	...	
696532	SL	83.3	hit_into_play	72.4	1.0	2595
696533	FF	98.9	foul	70.3	-31.0	2377
696534	FF	96.4	swinging_strike	0.0	0.0	2411
696535	FF	96.9	ball	0.0	0.0	2358
696536	FF	98.2	ball	0.0	0.0	2457

2018 rows × 7 columns

In [7]:
```
shohei = shohei[~shohei['description'].str.contains('ball|hit_by_pitch
shohei
```

Out[7]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rat
29965	FF	99.2	foul_tip	0.0	0.0	2095
29966	FS	91.8	foul	65.7	-48.0	1293
29967	SL	85.5	swinging_strike	0.0	0.0	2358
29968	FF	97.4	foul	75.8	42.0	2008
29970	SL	85.2	hit_into_play	68.1	-2.0	2148
...	...	...	...	...	...	
696528	FF	99.1	swinging_strike	0.0	0.0	2496
696529	FF	97.2	called_strike	0.0	0.0	2358
696532	SL	83.3	hit_into_play	72.4	1.0	2595
696533	FF	98.9	foul	70.3	-31.0	2377
696534	FF	96.4	swinging_strike	0.0	0.0	2411

1287 rows × 7 columns

In [8]:
```
category_mapping = {
    'swinging_strike': 1,
    'called_strike': 2,
    'swinging_strike_blocked': 3,
    'foul': 4,
    'foul_tip': 5,
    'hit_into_play': 6
}

# Replace the string values in the 'description' column with the corre
shohei['description'] = shohei['description'].replace(category_mapping
```

```
/var/folders/4v/_s6ktch56r3dgggt7wmn18dm0000gn/T/ipykernel_16172/3339
944763.py:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/panda
s-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.htm
l#returning-a-view-versus-a-copy)
  shohei['description'] = shohei['description'].replace(category_mapp
ing)
```

In [9]:
```
unique_pitch_types_list = shohei['pitch_type'].unique()
print("Unique pitch types:", unique_pitch_types_list)
```

```
print('Unique pitch types:', unique_pitch_types_list)
```

```
Unique pitch types: ['FF' 'FS' 'SL' 'FC' 'CU']
```

In [10]:
```
counts = shohei['pitch_type'].value_counts()
counts
```

Out[10]:
```
pitch_type
FF    550
SL    312
FS    220
FC    168
CU     37
Name: count, dtype: int64
```

In [11]:
```
shohei = shohei[shohei['pitch_type'] != 'CU']
shohei
```

Out[11]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rate
29965	FF	99.2	5	0.0	0.0	2095.0
29966	FS	91.8	4	65.7	-48.0	1293.0
29967	SL	85.5	1	0.0	0.0	2358.0
29968	FF	97.4	4	75.8	42.0	2008.0
29970	SL	85.2	6	68.1	-2.0	2148.0
...	...	...	...	...	...	...
696528	FF	99.1	1	0.0	0.0	2496.0
696529	FF	97.2	2	0.0	0.0	2358.0
696532	SL	83.3	6	72.4	1.0	2595.0
696533	FF	98.9	4	70.3	-31.0	2377.0
696534	FF	96.4	1	0.0	0.0	2411.0

1250 rows × 7 columns

In [12]:
```
pitching = {
    'FF': 1,
    'SL': 2,
    'FS': 3,
    'FC': 4,
}

# Replace the string values in the 'description' column with the corre
shohei['pitch_type'] = shohei['pitch_type'].replace(pitching)
```

```
shohei['pitch_type'] = shohei['pitch_type'].replace(pitching)
```

```
/var/folders/4v/_s6ktch56r3dgggt7wmn18dm0000gn/T/ipykernel_16172/198947711.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  shohei['pitch_type'] = shohei['pitch_type'].replace(pitching)
```

In [13]:
```
shohei
```

Out[13]:

	pitch_type	release_speed	description	launch_speed	launch_angle	release_spin_rate
29965	1	99.2	5	0.0	0.0	2095.0
29966	3	91.8	4	65.7	-48.0	1293.0
29967	2	85.5	1	0.0	0.0	2358.0
29968	1	97.4	4	75.8	42.0	2008.0
29970	2	85.2	6	68.1	-2.0	2148.0
...	...	...	...	...	...	...
696528	1	99.1	1	0.0	0.0	2496.0
696529	1	97.2	2	0.0	0.0	2358.0
696532	2	83.3	6	72.4	1.0	2595.0
696533	1	98.9	4	70.3	-31.0	2377.0
696534	1	96.4	1	0.0	0.0	2411.0

1250 rows × 7 columns

In [14]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
import numpy as np
```

In [66]:
```
# description
X = shohei.drop(['description'], axis=1)  # Features
y = shohei['description']
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
```

In [69]:
```
from sklearn.metrics import classification_report, accuracy_score, cor

rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)
rf_y_pred = rf_model.predict(X_test)

print("\nRandom Forest Classifier Report:")
print(classification_report(y_test, rf_y_pred))
print("Accuracy:", accuracy_score(y_test, rf_y_pred))


cm = confusion_matrix(y_test, rf_y_pred)

print("Confusion Matrix:")
print(cm)
```

```
Random Forest Classifier Report:
              precision    recall  f1-score   support

           1       0.43      0.38      0.41       143
           2       0.54      0.71      0.61       173
           3       0.33      0.06      0.10        18
           4       0.73      0.62      0.67       226
           5       0.00      0.00      0.00        11
           6       0.69      0.75      0.72       179

    accuracy                           0.61       750
   macro avg       0.46      0.42      0.42       750
weighted avg       0.60      0.61      0.60       750

Accuracy: 0.6053333333333333
Confusion Matrix:
[[ 55  81   2   4   1   0]
 [ 42 122   0   3   6   0]
 [ 13   4   1   0   0   0]
 [ 12  12   0 141   0  61]
 [  5   5   0   1   0   0]
 [  0   0   0  44   0 135]]
```

In [70]:
```
tpr = []
tnr = []
total_sum = np.sum(cm)

for i in range(cm.shape[0]):
    TP = cm[i, i]
    FN = np.sum(cm[i, :]) - TP
    FP = np.sum(cm[:, i]) - TP
    TN = total_sum - TP - FP - FN
```

```
    tpr.append(TP / (TP + FN) if TP + FN != 0 else 0)
    tnr.append(TN / (TN + FP) if TN + FP != 0 else 0)

rounded_tpr = [round(x, 2) for x in tpr]

print('TPR:', rounded_tpr)

rounded_tnr = [round(x, 2) for x in tnr]

print('TNR:',rounded_tnr)
```

```
TPR: [0.38, 0.71, 0.06, 0.62, 0.0, 0.75]
TNR: [0.88, 0.82, 1.0, 0.9, 0.99, 0.89]
```

Random Forest Classifier

- Accuracy: 60.5%
- Precision & Recall (TPR): Varies across classes; best recall for class 6 (0.77) and class 2 (0.69).
- TNR: Generally high, especially for class 3 (1.0) and class 5 (0.99).
- Observations: Random Forest shows a balanced performance across classes but struggles with class 3 and 5, possibly due to fewer instances of these classes.

In [18]:
```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()

# Train the model
log_reg.fit(X_train, y_train)
# Predictions
log_reg_y_pred = log_reg.predict(X_test)

# Evaluation
print("Logistic Regression Report:")
print(classification_report(y_test, log_reg_y_pred))

print("Accuracy Score:", accuracy_score(y_test, log_reg_y_pred))

cm = confusion_matrix(y_test, log_reg_y_pred)

print("Confusion Matrix:")
print(cm)
```

```
Logistic Regression Report:
              precision    recall  f1-score   support

           1       0.51      0.16      0.24       143
```

```
           2       0.51      0.95      0.66       173
           3       0.00      0.00      0.00        18
           4       0.76      0.65      0.70       226
           5       0.00      0.00      0.00        11
           6       0.71      0.74      0.73       179

    accuracy                           0.62       750
   macro avg       0.42      0.42      0.39       750
weighted avg       0.61      0.62      0.58       750

Accuracy Score: 0.624
Confusion Matrix:
[[ 23 120   0   0   0   0]
 [  8 165   0   0   0   0]
 [  9   9   0   0   0   0]
 [  3  22   0 148   0  53]
 [  2   9   0   0   0   0]
 [  0   0   0  47   0 132]]

/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/linea
r_model/_logistic.py:460: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as sho
wn in:
    https://scikit-learn.org/stable/modules/preprocessing.html (http
s://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver option
s:
    https://scikit-learn.org/stable/modules/linear_model.html#logisti
c-regression (https://scikit-learn.org/stable/modules/linear_model.ht
ml#logistic-regression)
  n_iter_i = _check_optimize_result(
/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [10]:

```
tpr = []
```

In [19]:
```
tpr = []
tnr = []
total_sum = np.sum(cm)

for i in range(cm.shape[0]):
    TP = cm[i, i]
    FN = np.sum(cm[i, :]) - TP
    FP = np.sum(cm[:, i]) - TP
    TN = total_sum - TP - FP - FN

    tpr.append(TP / (TP + FN) if TP + FN != 0 else 0)
    tnr.append(TN / (TN + FP) if TN + FP != 0 else 0)

rounded_tpr = [round(x, 2) for x in tpr]

print('TPR:', rounded_tpr)

rounded_tnr = [round(x, 2) for x in tnr]

print('TNR:',rounded_tnr)
```

```
TPR: [0.16, 0.95, 0.0, 0.65, 0.0, 0.74]
TNR: [0.96, 0.72, 1.0, 0.91, 1.0, 0.91]
```

Logistic Regression

- Accuracy: 62.4%
- Precision & Recall (TPR): High recall for class 2 (0.95) but very low for classes 1, 3, and 5.
- TNR: High for most classes, indicating good identification of negatives.
- Observations: Logistic Regression shows good overall accuracy but seems to be biased towards class 2, as indicated by its high recall.

In [20]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score, con


dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
dt_y_pred = dt_model.predict(X_test)

print("Decision Tree Classifier Report:")
print(classification_report(y_test, dt_y_pred))
print("Accuracy:", accuracy_score(y_test, dt_y_pred))


cm = confusion_matrix(y_test, dt_y_pred)

print("Confusion Matrix:")
```

```
print('Confusion Matrix:')
print(cm)
```

```
Decision Tree Classifier Report:
              precision    recall  f1-score   support

           1       0.39      0.36      0.37       143
           2       0.47      0.51      0.49       173
           3       0.22      0.11      0.15        18
           4       0.61      0.58      0.60       226
           5       0.00      0.00      0.00        11
           6       0.63      0.66      0.64       179

    accuracy                           0.52       750
   macro avg       0.39      0.37      0.38       750
weighted avg       0.52      0.52      0.52       750

Accuracy: 0.5226666666666666
Confusion Matrix:
[[ 51  73   4  12   3   0]
 [ 56  89   2  10  16   0]
 [  7   9   2   0   0   0]
 [ 11  12   1 132   0  70]
 [  6   5   0   0   0   0]
 [  0   0   0  61   0 118]]
```

In [21]:

```
tpr = []
tnr = []
total_sum = np.sum(cm)

for i in range(cm.shape[0]):
    TP = cm[i, i]
    FN = np.sum(cm[i, :]) - TP
    FP = np.sum(cm[:, i]) - TP
    TN = total_sum - TP - FP - FN

    tpr.append(TP / (TP + FN) if TP + FN != 0 else 0)
    tnr.append(TN / (TN + FP) if TN + FP != 0 else 0)

rounded_tpr = [round(x, 2) for x in tpr]

print('TPR:', rounded_tpr)

rounded_tnr = [round(x, 2) for x in tnr]

print('TNR:',rounded_tnr)
```

```
TPR: [0.36, 0.51, 0.11, 0.58, 0.0, 0.66]
TNR: [0.87, 0.83, 0.99, 0.84, 0.97, 0.88]
```

Decision Tree Classifier

- Accuracy: 52.1%
- Precision & Recall (TPR): More balanced recall across classes compared to other models, but still lower overall.
- TNR: Good across most classes but lower compared to Random Forest and Logistic Regression.
- Observations: Decision Tree presents a more evenly distributed performance across classes but with overall lower accuracy.

In [22]:
```
from sklearn.svm import SVC
svm_model = SVC()
svm_model.fit(X_train, y_train)
svm_y_pred = svm_model.predict(X_test)
print("SVM Classifier Report:")
print(classification_report(y_test, svm_y_pred))
print("Accuracy:", accuracy_score(y_test, svm_y_pred))
cm = confusion_matrix(y_test, svm_y_pred)
print("Confusion Matrix:")
print(cm)
```

```
SVM Classifier Report:
              precision    recall  f1-score   support

           1       0.00      0.00      0.00       143
           2       0.27      0.93      0.41       173
           3       0.00      0.00      0.00        18
           4       0.50      0.00      0.01       226
           5       0.00      0.00      0.00        11
           6       0.24      0.20      0.22       179

    accuracy                           0.26       750
   macro avg       0.17      0.19      0.11       750
weighted avg       0.27      0.26      0.15       750

Accuracy: 0.26266666666666666
Confusion Matrix:
[[  0 100   0   0   0  43]
 [  0 161   0   0   0  12]
 [  0   1   0   0   0  17]
 [  0 191   0   1   0  34]
 [  0   9   0   0   0   2]
 [  0 143   0   1   0  35]]

/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
```

```
  _warn_prf(average, modifier, msg_start, len(result))
/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/jasont/opt/anaconda3/lib/python3.9/site-packages/sklearn/metri
cs/_classification.py:1471: UndefinedMetricWarning: Precision and F-s
core are ill-defined and being set to 0.0 in labels with no predicted
samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [23]:
```
tpr = []
tnr = []
total_sum = np.sum(cm)

for i in range(cm.shape[0]):
    TP = cm[i, i]
    FN = np.sum(cm[i, :]) - TP
    FP = np.sum(cm[:, i]) - TP
    TN = total_sum - TP - FP - FN

    tpr.append(TP / (TP + FN) if TP + FN != 0 else 0)
    tnr.append(TN / (TN + FP) if TN + FP != 0 else 0)

rounded_tpr = [round(x, 2) for x in tpr]

print('TPR:', rounded_tpr)

rounded_tnr = [round(x, 2) for x in tnr]

print('TNR:',rounded_tnr)
```

```
TPR: [0.0, 0.93, 0.0, 0.0, 0.0, 0.2]
TNR: [1.0, 0.23, 1.0, 1.0, 1.0, 0.81]
```

SVM Classifier

- Accuracy: 26.3%
- Precision & Recall (TPR): Extremely low performance across most classes, with a noticeable bias towards class 2.
- TNR: High for class 1, 3, and 5, but these are also classes with very low recall.
- Observations: SVM performs poorly in this scenario, with significant biases and overall low accuracy.

General Analysis

- Random Forest and Logistic Regression are the better performers in terms of accuracy. Random Forest shows a more balanced classification across different classes, while Logistic Regression seems biased towards certain classes.
- Decision Tree shows moderate performance with more evenly distributed recall rates.
- SVM underperforms significantly in this particular case.
- Models might be struggling with certain classes due to class imbalance or lack of sufficient representative data for those classes.
- The high TNR across models suggests they are good at identifying true negatives, but this should be weighed against their ability to correctly identify true positives (recall).

Recommendations

- Investigate class imbalance: Models might benefit from techniques like resampling or using class weights to handle classes with fewer instances.
- Feature Engineering: Explore if additional or different feature engineering can improve model performance.
- Hyperparameter Tuning: Each model may have specific parameters that can be tuned for better performance.
- Model Selection: Consider Random Forest or Logistic Regression for further tuning and use, as they show the most promise in this analysis.
- SVM's performance suggests it might not be suitable for this dataset as configured, or it may require significant hyperparameter tuning and feature scaling.

In [73]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRe
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import learning_curve
import scipy.stats as stats
```

In [74]:
```
# launch_speed
X = shohei.drop(['launch_speed','description'], axis=1)  # Features
y = shohei['launch_speed']  # Target variable

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
```

In [75]:
```
# Create a Linear Regression model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Make predictions
lr_predictions = lr_model.predict(X_test)
```

```
lr_predictions = lr_model.predict(X_test)

# Evaluate the model
lr_mse = mean_squared_error(y_test, lr_predictions)
print(f'Linear Regression MSE: {lr_mse}')
```

```
Linear Regression MSE: 1450.2217119294846
```

In [76]:
```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
lr_rmse = mean_squared_error(y_test, lr_predictions, squared=False)
lr_mae = mean_absolute_error(y_test, lr_predictions)
lr_r_squared = r2_score(y_test, lr_predictions)
print(f'Linear Regression RMSE: {lr_rmse}')
print(f'Linear Regression MAE: {lr_mae}')
print(f'Linear Regression R Square: {lr_r_squared}')
```

```
Linear Regression RMSE: 38.081776638301484
Linear Regression MAE: 34.55212387002685
Linear Regression R Square: 0.14196895916644459
```

In [77]:
```
n = X_test.shape[0]
p = X_test.shape[1]

lr_adjusted_r_squared = 1 - (1 - lr_r_squared) * (n - 1) / (n - p - 1)
print(f'Linear Regression Adjusted R-Squared: {lr_adjusted_r_squared}'
```

```
Linear Regression Adjusted R-Squared: 0.12438635587067493
```

In [78]:
```
lr_coefficients = lr_model.coef_

print("Coefficients for Linear Regression:")
for name, coef in zip(X_train.columns, lr_coefficients):
    print(f"{name}: {coef}")
```

```
Coefficients for Linear Regression:
pitch_type: 2.9729915206659916
release_speed: 1.0132054238255537
launch_angle: 0.5918365599306861
release_spin_rate: 0.0034611829023358798
release_extension: -26.277120403547638
```

Model Performance Metrics

- MSE: A high MSE of 1450.22 indicates that the model's predictions are, on average, quite far off from the actual values when considering the squared errors.
- RMSE: The RMSE of 38.08 provides an average error in the same units as the target variable, suggesting that some predictions could be off by approximately 38 units.
- MAE: An MAE of 34.55 suggests that the average absolute error is also quite large.

- R-Squared: A low R-squared value of 0.1419 indicates that the model explains only about 14.19% of the variability in the response variable.
- Adjusted R-Squared: The Adjusted R-squared of 0.1244 is even lower, accounting for the number of predictors, which further confirms that the model does not fit the data well.

In [79]:
```
lr_model.fit(X_train, y_train)

# Retrieve the coefficients and intercept
coefficients = lr_model.coef_
intercept = lr_model.intercept_

# Get the feature names from X_train
feature_names = X_train.columns

# Print the linear regression equation
print("Linear Regression Equation:")
print(f"launch_speed = {intercept}", end=" ")
for coef, name in zip(coefficients, feature_names):
    print(f"+ ({coef})*{name}", end=" ")
```

```
Linear Regression Equation:
launch_speed = 110.33867546622845 + (2.9729915206659916)*pitch_type +
(1.0132054238255537)*release_speed + (0.5918365599306861)*launch_angl
e + (0.0034611829023358798)*release_spin_rate + (-26.27712040354763
8)*release_extension
```

In [80]:
```
lr_cv_scores = cross_val_score(lr_model, X, y, cv=5, scoring='neg_mean
lr_cv_scores_mean = lr_cv_scores.mean()
print(f'Linear Regression - Mean Cross-Validated MSE: {lr_cv_scores_me
```

```
Linear Regression - Mean Cross-Validated MSE: -1557.495440467538
```

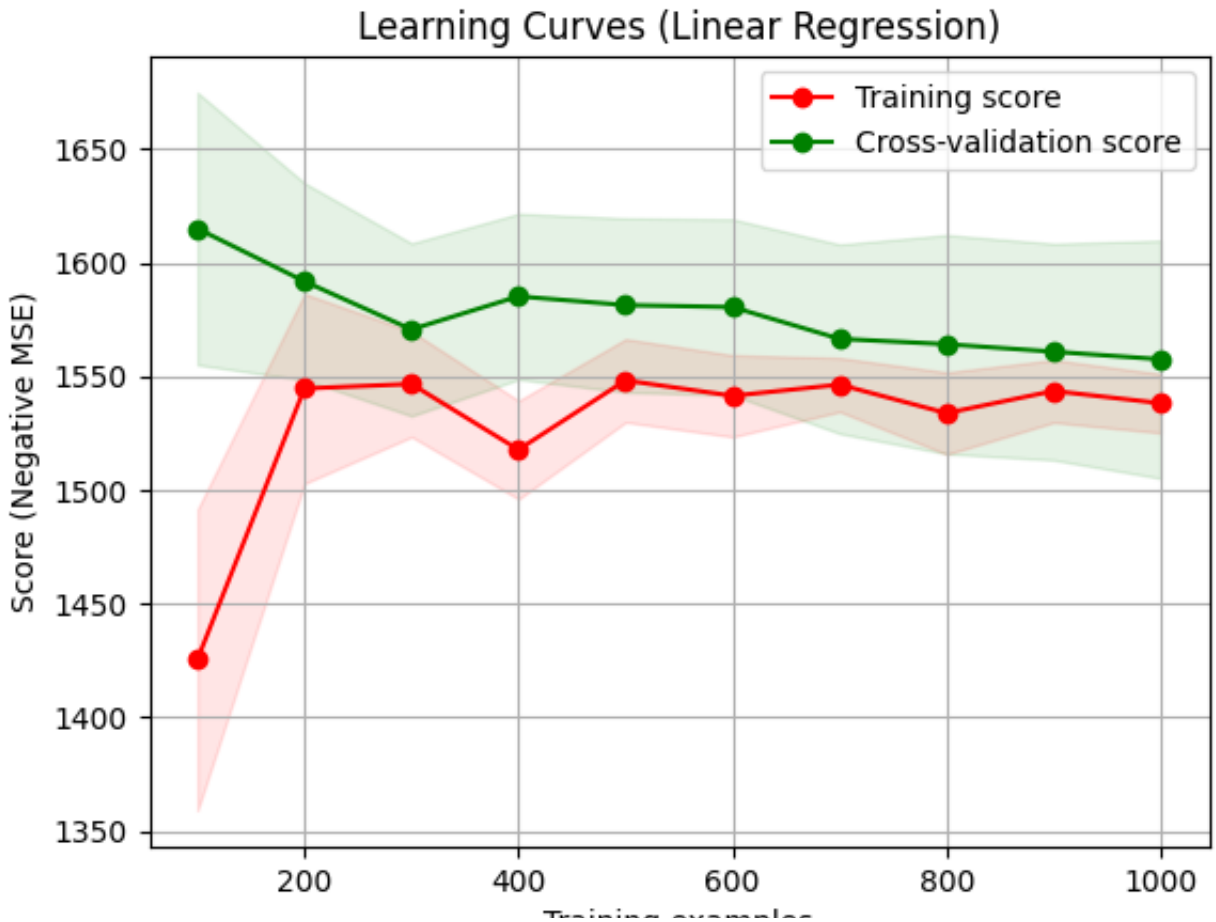
Cross-Validated MSE

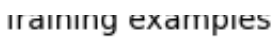
- Cross-Validated MSE: A cross-validated MSE of -1557.495 suggests that the model is not performing well across different subsets of the data, reinforcing the conclusions from the R-squared values.

In [81]:
```
def plot_learning_curves(model, X, y, title):
    train_sizes, train_scores, test_scores = learning_curve(model, X,
                                                            train_size
                                                            scoring="n


    # Mean and Standard Deviation of training set scores
    train_scores_mean = -train_scores.mean(axis=1)
```

```
    train_scores_std = train_scores.std(axis=1)

    # Mean and Standard Deviation of test set scores
    test_scores_mean = -test_scores.mean(axis=1)
    test_scores_std = test_scores.std(axis=1)

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std
                     train_scores_mean + train_scores_std, alpha=0.1,
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, co
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="T
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cr

    plt.title(title)
    plt.xlabel("Training examples")
    plt.ylabel("Score (Negative MSE)")
    plt.legend(loc="best")
    plt.grid()
    plt.show()
```

In [82]:
```
import matplotlib.pyplot as plt
lr_model = LinearRegression()
plot_learning_curves(lr_model, X, y, "Learning Curves (Linear Regressi
```

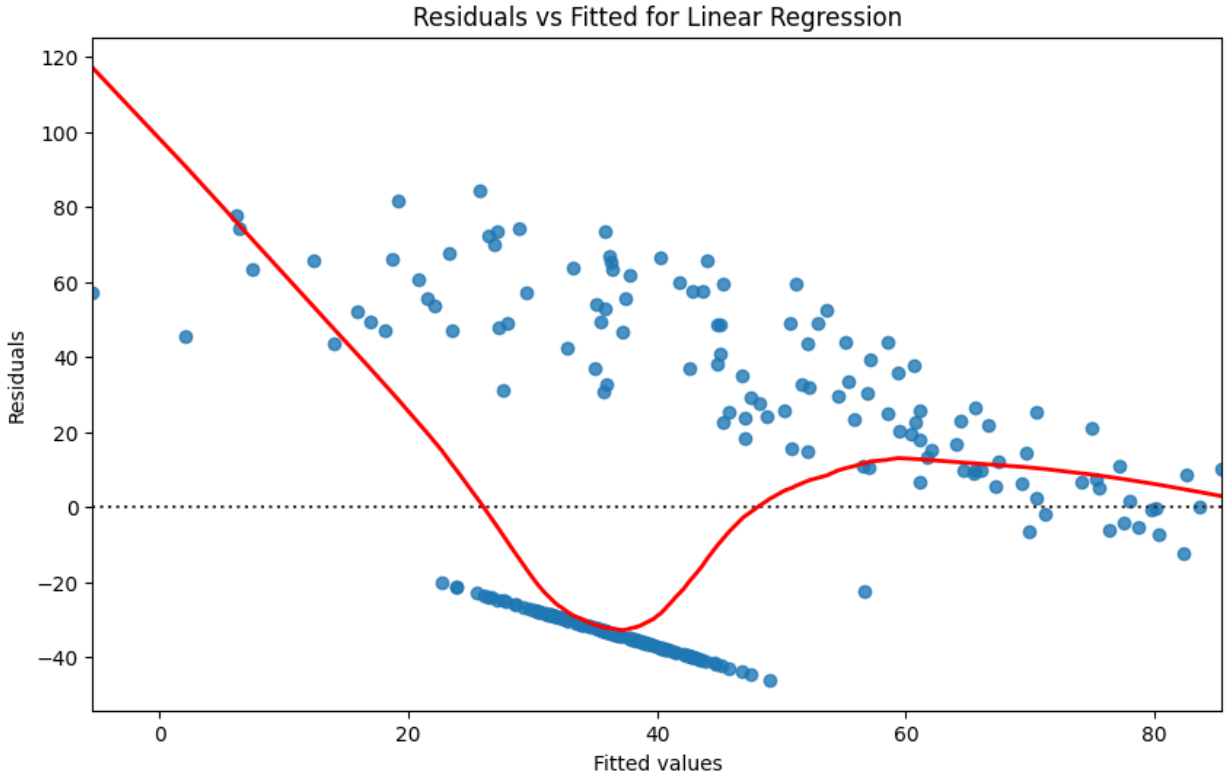




Learning Curves for Linear Regression

- Training Score: The training score starts high and decreases as more data is added, which is typical as the model begins to encounter more variance in the data.
- Cross-Validation Score: The cross-validation score starts low and increases, suggesting that as the model is trained on more data, it is better able to generalize. However, it still shows a high negative MSE, indicating errors are relatively large.
- Gap Between Scores: The gap between the training and cross-validation scores narrows with more data, but does not close entirely, suggesting that the model may benefit from further optimization, more data, or may have some inherent limitations in capturing the dataset's complexity.

In [83]:
```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(10, 6))
sns.residplot(x=lr_predictions, y=y_test, lowess=True, line_kws={'colo
plt.title('Residuals vs Fitted for Linear Regression')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
plt.show()
```



Residuals vs Fitted for Linear Regression

- Residual Pattern: The residuals are not randomly scattered around the horizontal line at zero. Instead, they exhibit a clear pattern, which suggests that the linear model may not be capturing some non-linear aspects of the relationship between the variables.
- Homoscedasticity: The residuals do not show constant variance (homoscedasticity). The variance of residuals appears to increase as the fitted values increase, which is a sign of heteroscedasticity.

In [84]:
```
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions
rf_predictions = rf_model.predict(X_test)

# Evaluate the model
rf_mse = mean_squared_error(y_test, rf_predictions)
print(f'Random Forest MSE: {rf_mse}')
```

```
Random Forest MSE: 155.425248316
```

In [85]:
```
rf_rmse = mean_squared_error(y_test, rf_predictions, squared=False)
rf_mae = mean_absolute_error(y_test, rf_predictions)
rf_r_squared = r2_score(y_test, rf_predictions)
print(f'Random Forest Regressor RMSE: {rf_mae}')
print(f'Random Forest Regressor MAE: {rf_mae}')
print(f'Random Forest Regressor R Square: {rf_r_squared}')
```

```
Random Forest Regressor RMSE: 6.622131999999999
Random Forest Regressor MAE: 6.622131999999999
Random Forest Regressor R Square: 0.9080418625046239
```

In [86]:
```
n = X_test.shape[0]
p = X_test.shape[1]

rf_adjusted_r_squared = 1 - (1 - rf_r_squared) * (n - 1) / (n - p - 1)
print(f'Random Forest Regressor Adjusted R-Squared: {rf_adjusted_r_squ
```

```
Random Forest Regressor Adjusted R-Squared: 0.9061574744411941
```

Model Performance Metrics

- MSE: A low MSE of 155.42 suggests the model's predictions are fairly accurate on average, with a small squared error.
- RMSE: The RMSE of 6.62, which gives the standard deviation of the model's prediction errors, indicates that the average prediction error is relatively low.
- MAE: MAE which should generally be lower than the RMSE.
- R-Squared: A high R-squared value of 0.9080 suggests the model explains a significant

- R-Squared: A high R-squared value of 0.9060 suggests the model explains a significant amount of the variance in the target variable.
- Adjusted R-Squared: The Adjusted R-squared of 0.9061 is very close to the R-squared value, which is excellent, especially since it adjusts for the number of predictors in the model.

In [87]:
```
# Feature importance
rf_feature_importance = rf_model.feature_importances_

# Print feature importance
print("Feature Importances for Random Forest Regressor:")
for name, importance in zip(X_train.columns, rf_feature_importance):
    print(f"{name}: {importance}")
```

```
Feature Importances for Random Forest Regressor:
pitch_type: 0.0022121354559491995
release_speed: 0.021100313803641593
launch_angle: 0.9388307113738563
release_spin_rate: 0.030833082799748078
release_extension: 0.0070237565668048365
```

Feature Importances for Random Forest Regressor

- Feature Importance: The launch_angle has the highest importance, suggesting it is the most significant predictor for the model. This should be interpreted with caution, as high importance might be due to the scale of the feature or its variance rather than its actual predictive power.

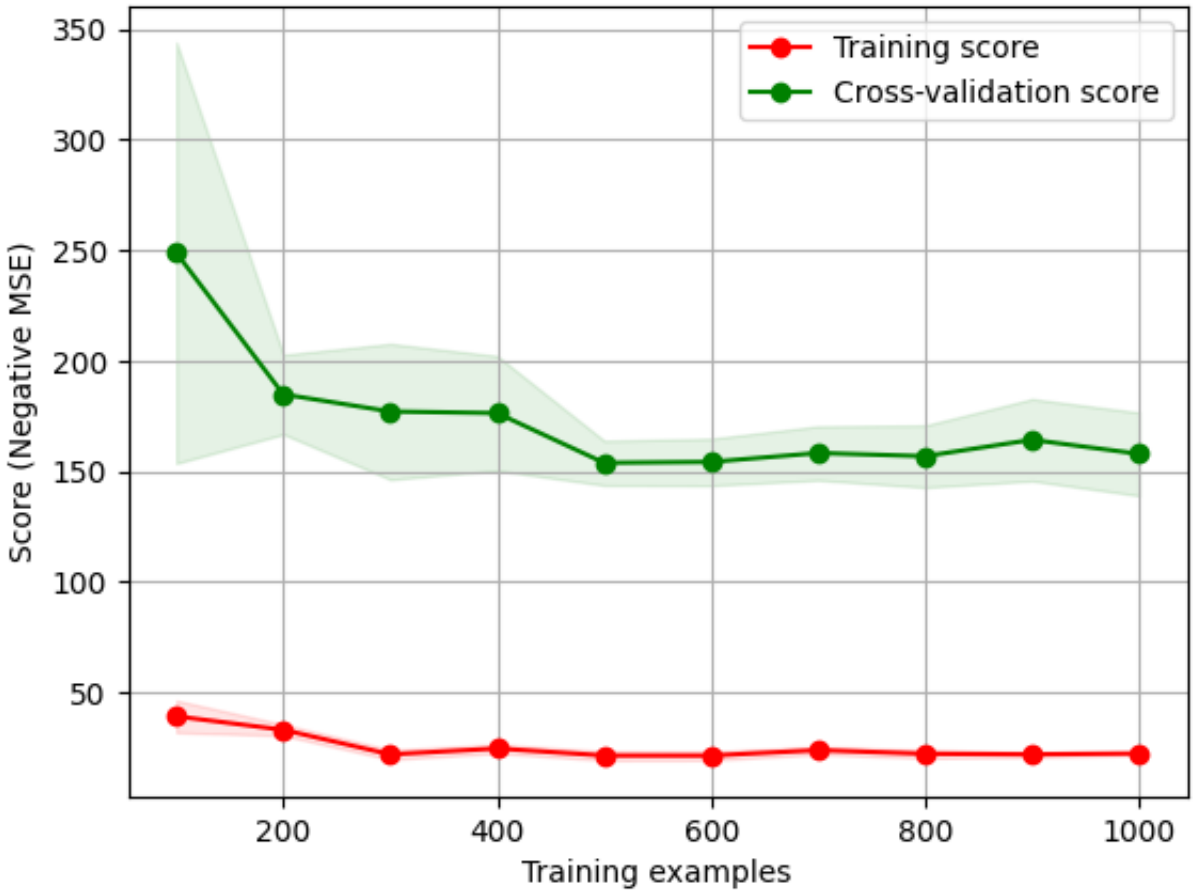
In [88]:
```
rf_cv_scores = cross_val_score(rf_model, X, y, cv=5, scoring='neg_mean
rf_cv_scores_mean = rf_cv_scores.mean()
print(f'Random Forest Regressor - Mean Cross-Validated MSE: {rf_cv_sco
```

```
Random Forest Regressor - Mean Cross-Validated MSE: -156.514999355200
05
```

Cross-Validated MSE

- Cross-Validated MSE: The negative MSE from cross-validation is consistent with the standalone MSE, further validating the model's performance. The negative sign is due to scikit-learn's conventions and should be considered as a positive value for MSE.

In [89]:
```
rf_model = RandomForestRegressor()
plot_learning_curves(rf_model, X, y, "Learning Curves (Random Forest R
```

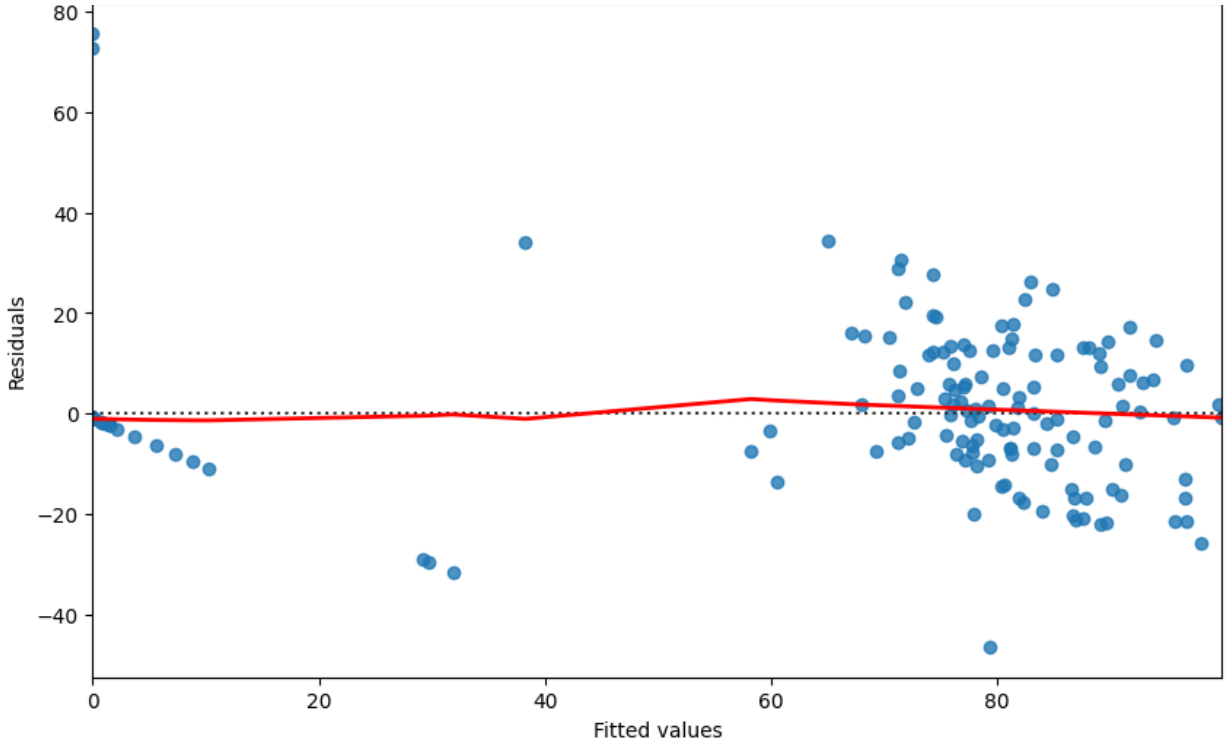


Learning Curves for Random Forest Regressor

- Training Score (Red Line): The training score is relatively low at the start and improves slightly as more training examples are used. This suggests that the model is able to learn from the data effectively, and additional data helps improve its accuracy.
- Cross-Validation Score (Green Line): The cross-validation score starts higher and decreases as more data is added, indicating that the model's ability to generalize improves with more data. The scores converge as the number of training examples increases, which suggests a good fit without overfitting.
- Stability of Scores: Both training and validation scores remain relatively stable across the number of training examples, which implies the model is stable and additional data may not significantly change performance.

In [90]:
```
plt.figure(figsize=(10, 6))
sns.residplot(x=rf_predictions, y=y_test, lowess=True, line_kws={'colo
plt.title('Residuals vs Fitted for Random Forest Regressor')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
plt.show()
```

Residuals vs Fitted for Random Forest Regressor



Residuals vs Fitted for Random Forest Regressor

- Residuals Distribution: The residuals are distributed around the horizontal line at zero without any clear pattern, which is a good sign as it suggests that the model's predictions do not have systematic errors across the range of predictions.
- Consistency: There's no obvious structure or trend in the residuals, indicating that the model's errors are consistent across different levels of fitted values.

Conclusion and Recommendations

- The Random Forest Regressor appears to be a good fit for the data, with a high R-squared value and low MSE and RMSE.
- The residual plot does not show any concerning patterns that would indicate the model is not capturing the underlying data structure.
- The learning curves suggest that the model is stable and generalizes well, with no signs of overfitting.
- Considering the high importance of launch_angle, it might be useful to further investigate this feature's relationship with the target variable and consider any necessary feature engineering.
- The model's robustness, as indicated by the cross-validation and learning curve analysis, suggests it would perform well on unseen data.

In [91]:
```
# Create a Gradient Boosting Regressor model
```

```
gb_model = GradientBoostingRegressor(random_state=42)
gb_model.fit(X_train, y_train)

# Make predictions
gb_predictions = gb_model.predict(X_test)

# Evaluate the model
gb_mse = mean_squared_error(y_test, gb_predictions)
print(f'Gradient Boosting MSE: {gb_mse}')
```

```
Gradient Boosting MSE: 142.163180257569
```

In [92]:
```
gb_rmse = mean_squared_error(y_test, gb_predictions, squared=False)
gb_mae = mean_absolute_error(y_test, gb_predictions)
gb_r_squared = r2_score(y_test, gb_predictions)
print(f'Gradient Boosting Regressor RMSE: {gb_rmse}')
print(f'Gradient Boosting Regressor MAE: {gb_mae}')
print(f'Gradient Boosting Regressor R Square: {gb_r_squared}')
```

```
Gradient Boosting Regressor RMSE: 11.9232202134142
Gradient Boosting Regressor MAE: 6.710799493157916
Gradient Boosting Regressor R Square: 0.9158884324229857
```

In [43]:
```
n = X_test.shape[0]
p = X_test.shape[1]

gb_adjusted_r_squared = 1 - (1 - gb_r_squared) * (n - 1) / (n - p - 1)
print(f'Gradient Boosting Regressor Adjusted R-Squared: {gb_adjusted_r
```

```
Gradient Boosting Regressor Adjusted R-Squared: 0.9141648347267354
```

Model Performance Metrics

- MSE: A MSE of 142.16 suggests the model has an average squared prediction error of this magnitude, which is relatively low.
- RMSE: An RMSE of 11.92, which is the square root of the MSE, indicates the standard deviation of the residuals. The RMSE is in the same units as the predicted quantity, making it easier to interpret.
- MAE: An MAE of 6.71 is lower than the RMSE, as expected, because MAE is less sensitive to large errors than RMSE.
- R-Squared: A high R-squared value of 0.9159 means the model explains approximately 91.59% of the variance in the target variable, which is excellent.
- Adjusted R-Squared: The Adjusted R-squared value is very close to the R-squared value, indicating that the number of predictors is appropriate for the amount of data and the model is not being penalized for unnecessary complexity.

In [44]:
```
gb_feature_importance = gb_model.feature_importances_
```

```
# Print feature importance
print("Feature Importances for Gradient Boosting Regressor:")
for name, importance in zip(X_train.columns, gb_feature_importance):
    print(f"{name}: {importance}")
```

```
Feature Importances for Gradient Boosting Regressor:
pitch_type: 0.0010592306131779964
release_speed: 0.009962242633953483
launch_angle: 0.971051838107468
release_spin_rate: 0.01582145618773325
release_extension: 0.0021052324576672178
```

Feature Importances for Gradient Boosting Regressor

- Feature Importance: The model gives overwhelming importance to launch_angle, which indicates that this feature plays a critical role in predicting the outcome. This could be expected if the launch angle is indeed a strong determinant of the response variable in your specific context.

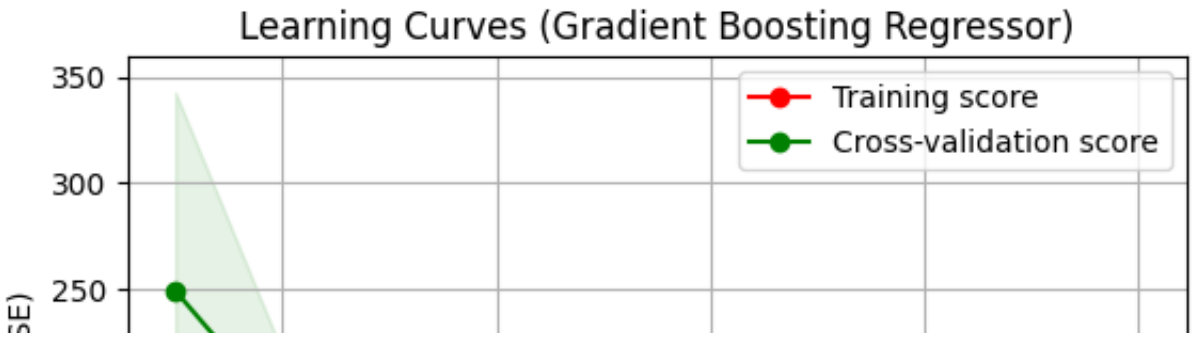
In [45]:
```
gb_cv_scores = cross_val_score(gb_model, X, y, cv=5, scoring='neg_mean
gb_cv_scores_mean = gb_cv_scores.mean()
print(f'Gradient Boosting Regressor - Mean Cross-Validated MSE: {gb_cv
```

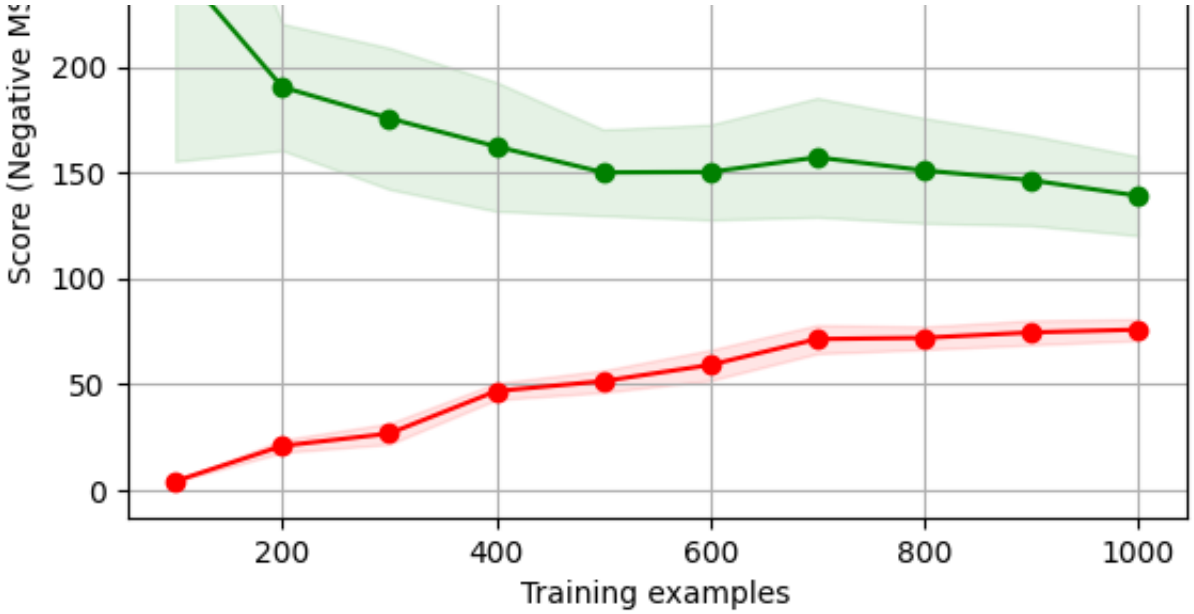
```
Gradient Boosting Regressor - Mean Cross-Validated MSE: -138.95797869
48027
```

Cross-Validated MSE

- Cross-Validated MSE: A negative cross-validated MSE of -138.96, which would be positive when considering the absolute value, confirms the standalone MSE. This suggests that the model's good performance is consistent across different subsets of the data.

In [46]:
```
gb_model = GradientBoostingRegressor()
plot_learning_curves(gb_model, X, y, "Learning Curves (Gradient Boosti
```

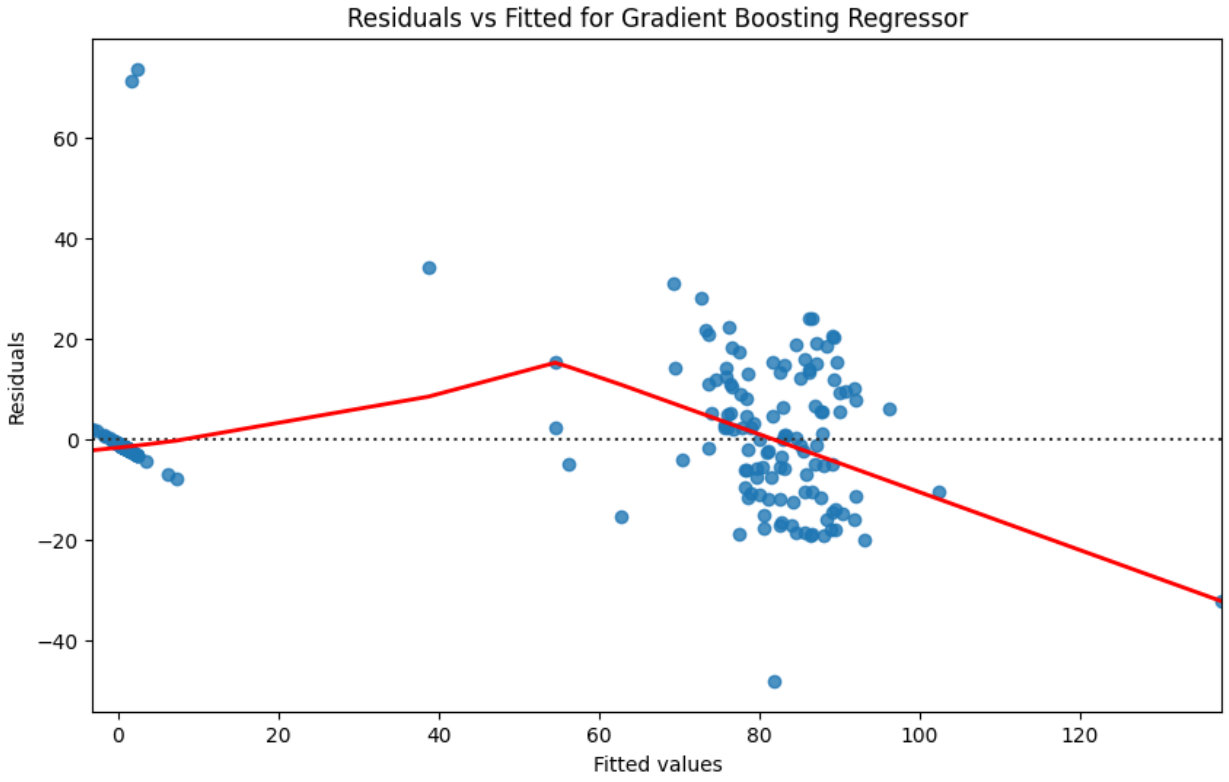




Learning Curves for Gradient Boosting Regressor

- Training Score (Red Line): The training score starts at a relatively high negative MSE, indicating the model is underfitting with very few training examples. However, as more training data is added, the training score improves significantly, suggesting that the model is learning effectively.
- Cross-Validation Score (Green Line): The cross-validation score also starts high (negative MSE) but improves as more training data is provided. It remains relatively stable after a certain point, indicating that adding more data doesn't significantly improve the model's ability to generalize.
- Gap Between Scores: The gap between training and cross-validation scores narrows with more data, suggesting a good balance between the model's ability to fit the training data and generalize to unseen data. The gap does not close entirely, which is typical and acceptable in practice.

In [47]:
```
plt.figure(figsize=(10, 6))
sns.residplot(x=gb_predictions, y=y_test, lowess=True, line_kws={'colo
plt.title('Residuals vs Fitted for Gradient Boosting Regressor')
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
plt.show()
```



Residuals vs Fitted for Gradient Boosting Regressor

- Residuals Distribution: The residuals should be randomly distributed around the horizontal line at zero if the model is appropriately fitted. In the plot, there seems to be no clear pattern or systematic bias in the residuals, which is a good indication.
- Potential Non-Linearity: There is a slight curve in the residuals, which might indicate some non-linear relationships that the model is not capturing entirely. This could be addressed by adding non-linear transformations of features or interaction terms.

Conclusion and Recommendations

- The Gradient Boosting Regressor seems to be an effective model for the dataset, with high predictive performance and generalization capability as indicated by the R-squared values and learning curves.
- The importance of launch_angle could suggest a strong reliance on this feature, which might be good if it's justified by the domain knowledge. However, it also implies that the model could be sensitive to changes in this particular feature.
- The residual plot's slight curve warrants further investigation into potential non-linear relationships.
- Cross-validation confirms the model's robustness, suggesting it would perform well on unseen data.

Error Metrics

1. Mean Squared Error (MSE): This represents the average of the squares of the errors or deviations. Lower values are better.
    - Linear Regression: 1450.22
    - Random Forest: 155.43
    - Gradient Boosting: 142.16
2. Root Mean Squared Error (RMSE): This is the square root of MSE and provides a measure of the magnitude of the error. Lower values are better.
    - Linear Regression: 38.08
    - Random Forest: 6.62
    - Gradient Boosting: 11.92
3. Mean Absolute Error (MAE): This is the average of the absolute errors and is less sensitive to outliers than MSE. Lower values are better.
    - Linear Regression: 34.55
    - Random Forest: 6.62
    - Gradient Boosting: 6.71

R-squared and Adjusted R-squared

- R-squared: Indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. Higher is better.
- Adjusted R-squared: Adjusts R-squared for the number of predictors in the model; more useful for comparing models with a different number of predictors.
  - Linear Regression: $R^2 = 0.142$, Adjusted $R^2 = 0.124$
  - Random Forest: $R^2 = 0.908$, Adjusted $R^2 = 0.906$
  - Gradient Boosting: $R^2 = 0.916$, Adjusted $R^2 = 0.914$

Feature Importance

- Indicates how important each feature is for the prediction. Higher values mean more importance.
  - For both Random Forest and Gradient Boosting, `launch_angle` is the most significant predictor.

Cross-Validated MSE

- This is the MSE obtained from cross-validation and helps to assess the model's performance on unseen data.
  - Linear Regression: -1557.50
  - Random Forest: -156.51
  - Gradient Boosting: -138.96 (better performance indicated by a higher negative value)

Summary

- Random Forest and Gradient Boosting significantly outperform Linear Regression in this scenario, evidenced by much lower MSE, RMSE, MAE, and higher R-squared values.
- Gradient Boosting appears to be the best model overall, with the lowest MSE and the highest R-squared values, suggesting it's the most accurate and best at explaining the variance in your data.
- The feature importances indicate that `launch_angle` is a critical predictor in both the Random Forest and Gradient Boosting models.
- The negative cross-validated MSE values are higher (less negative) for the Random Forest and Gradient Boosting models, indicating better performance in cross-validation compared to Linear Regression.