**Comparing Outlier Detection Methods**

Using batting stats from Major League Baseball’s 2023 season

[[John Andrews](https://medium.com/@andrews_or?source=post_page-----956f4b097061--------------------------------)](https://medium.com/@andrews_or?source=post_page-----956f4b097061--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----956f4b097061--------------------------------)](https://towardsdatascience.com/?source=post_page-----956f4b097061--------------------------------)

[John Andrews](https://medium.com/@andrews_or?source=post_page-----956f4b097061--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F58a322305d14&operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fcomparing-outlier-detection-methods-956f4b097061&user=John+Andrews&userId=58a322305d14&source=post_page-58a322305d14----956f4b097061---------------------post_header-----------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----956f4b097061--------------------------------)

·

12 min read

·

2 days ago

127

2



Shohei Ohtani, photo by [Erik Drost](https://www.flickr.com/photos/edrost88/) on [Flikr](https://www.flickr.com/photos/edrost88/48484060286/in/album-72157629239820345/), [CC BY 2.0](https://commons.wikimedia.org/w/index.php?curid=88998688)

*Outlier detection* is an unsupervised machine learning task to identify anomalies (unusual observations) within a given data set. This task is helpful in many real-world cases where our available dataset is already “contaminated” by anomalies. Scikit-learn [implements several outlier detection algorithms](https://scikit-learn.org/stable/modules/outlier_detection.html), and in cases where we have an *uncontaminated* baseline, we can also use these algorithms for *novelty detection*, a semi-supervised task that predicts whether new observations are outliers.

**Overview**

The four outlier detection algorithms we’ll compare are:

* [Elliptic Envelope](https://scikit-learn.org/stable/modules/generated/sklearn.covariance.EllipticEnvelope.html) is suitable for normally-distributed data with low dimensionality. As its name implies, it uses the multivariate normal distribution to create a distance measure to separate outliers from inliers.
* [Local Outlier Factor](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html) is a comparison of the local density of an observation with that of its neighbors. Observations with much lower density than their neighbors are considered outliers.
* [One-Class Support Vector Machine (SVM) with Stochastic Gradient Descent (SGD)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDOneClassSVM.html) is an O(n) approximate solution of the One-Class SVM. Note that the O(n²) [One-Class SVM](https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html#sklearn.svm.OneClassSVM) works well on our small example dataset but may be impractical for your actual use case.
* [Isolation Forest](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html) is a tree-based approach where outliers are more quickly isolated by random splits than inliers.

Since our task is unsupervised, we don’t have ground truth to compare accuracies of these algorithms. Instead, we want to see how their results (player rankings in particular) differ from one another and gain some intuition into their behavior and limitations, so that we might know when to prefer one over another.

Let’s compare a few of these techniques using two metrics of batter performance from 2023’s Major Leage Baseball (MLB) season:

* On-base percentage (OBP), the rate at which a batter reaches base (by hitting, walking, or getting hit by pitch) per plate appearance
* Slugging (SLG), the average number of total bases per at bat

There are [many more sophisticated metrics of batter performance](https://library.fangraphs.com/offense/offensive-statistics-list/), including OBP plus SLG (OPS), weighted on-base average (wOBA), and adjusted weighted runs created (WRC+). However, we’ll see that in addition to being [commonly used](https://www.mlb.com/glossary/miscellaneous/slash-line) and easy to understand, OBP and SLG are moderately correlated and approximately normally distributed, making them well suited for this comparison.

**Dataset preparation**

We use the pybaseball package to obtain hitting data. This Python package is under MIT [license](https://github.com/jldbc/pybaseball/blob/master/LICENSE) and returns data from [Fangraphs.com](https://www.fangraphs.com/leaders/major-league?pos=all&stats=bat&lg=all&qual=y&type=8&season=2023&month=0&season1=2023&ind=0), [Baseball-Reference.com](https://www.baseball-reference.com/leagues/majors/2023-standard-batting.shtml), and other sources which have in turn obtained offical records from Major League Baseball.

We use pybaseball’s 2023 batting statistics, which can be obtained either by batting\_stats (FanGraphs) or batting\_stats\_bref (Baseball Reference). It turns out that the player names [are more correctly formatted](https://github.com/jldbc/pybaseball/issues/393) from Fangraphs, but player teams and leagues from Baseball Reference are better formatted in the case of traded players. For a dataset with improved readability, we actually need to merge three tables: FanGraphs, Baseball Reference, and a key lookup.

from pybaseball import (cache, batting\_stats\_bref, batting\_stats,   
 playerid\_reverse\_lookup)  
import pandas as pd  
  
cache.enable() # avoid unnecessary requests when re-running  
  
MIN\_PLATE\_APPEARANCES = 200  
  
# For readability and reasonable default sort order  
df\_bref = batting\_stats\_bref(2023).query(f"PA >= {MIN\_PLATE\_APPEARANCES}"  
 ).rename(columns={"Lev":"League",  
 "Tm":"Team"}  
 )  
df\_bref["League"] = \  
 df\_bref["League"].str.replace("Maj-","").replace("AL,NL","NL/AL"  
 ).astype('category')  
  
df\_fg = batting\_stats(2023, qual=MIN\_PLATE\_APPEARANCES)  
  
key\_mapping = \  
 playerid\_reverse\_lookup(df\_bref["mlbID"].to\_list(), key\_type='mlbam'  
 )[["key\_mlbam","key\_fangraphs"]  
 ].rename(columns={"key\_mlbam":"mlbID",  
 "key\_fangraphs":"IDfg"}  
 )  
  
df = df\_fg.drop(columns="Team"  
 ).merge(key\_mapping, how="inner", on="IDfg"  
 ).merge(df\_bref[["mlbID","League","Team"]],  
 how="inner", on="mlbID"  
 ).sort\_values(["League","Team","Name"])

**Data Exploration**

First, we note that these metrics differ in mean and variance and are moderately correlated. We also note that each metric is fairly symmetric, with median value close to mean.

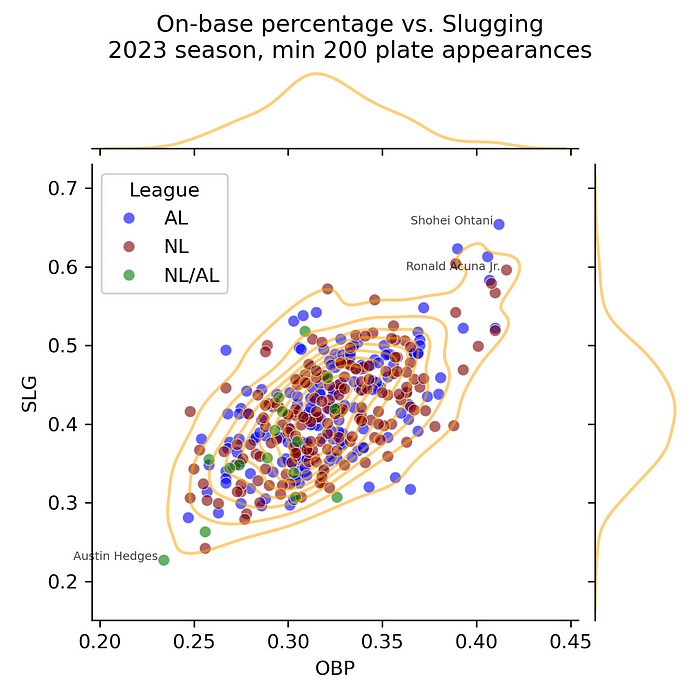
print(df[["OBP","SLG"]].describe().round(3))  
  
print(f"\nCorrelation: {df[['OBP','SLG']].corr()['SLG']['OBP']:.3f}")

OBP SLG  
count 362.000 362.000  
mean 0.320 0.415  
std 0.034 0.068  
min 0.234 0.227  
25% 0.300 0.367  
50% 0.318 0.414  
75% 0.340 0.460  
max 0.416 0.654  
  
Correlation: 0.630

Let’s visualize this joint distribution, using:

* Scatterplot of the players, colored by National League (NL) vs American League (AL)
* Bivariate kernel density estimator (KDE) plot of the players, which smoothes the scatterplot with a Gaussian kernel to estimate density
* Marginal KDE plots of each metric

import matplotlib.pyplot as plt  
import seaborn as sns  
  
g = sns.JointGrid(data=df, x="OBP", y="SLG", height=5)  
g = g.plot\_joint(func=sns.scatterplot, data=df, hue="League",  
 palette={"AL":"blue","NL":"maroon","NL/AL":"green"},  
 alpha=0.6  
 )  
g.fig.suptitle("On-base percentage vs. Slugging\n2023 season, min "  
 f"{MIN\_PLATE\_APPEARANCES} plate appearances"  
 )  
g.figure.subplots\_adjust(top=0.9)  
sns.kdeplot(x=df["OBP"], color="orange", ax=g.ax\_marg\_x, alpha=0.5)  
sns.kdeplot(y=df["SLG"], color="orange", ax=g.ax\_marg\_y, alpha=0.5)  
sns.kdeplot(data=df, x="OBP", y="SLG",  
 ax=g.ax\_joint, color="orange", alpha=0.5  
 )  
df\_extremes = df[ df["OBP"].isin([df["OBP"].min(),df["OBP"].max()])   
 | df["OPS"].isin([df["OPS"].min(),df["OPS"].max()])  
 ]  
  
for \_,row in df\_extremes.iterrows():  
 g.ax\_joint.annotate(row["Name"], (row["OBP"], row["SLG"]),size=6,  
 xycoords='data', xytext=(-3, 0),  
 textcoords='offset points', ha="right",  
 alpha=0.7)  
plt.show()



The top-right corner of the scatterplot shows a cluster of excellence in hitting corresponding to the heavy upper tails of the SLG and OBP distributions. This small group excels at getting on base *and* hitting for extra bases. How much we consider them to be outliers (because of their distance from the majority of the player population) versus inliers (because of their proximity to one another) depends on the definition used by our selected algorithm.

**Apply outlier detection algorithms**

Scikit-learn’s outlier detection algorithms typically have fit() and predict() methods, but there are exceptions and also differences between algorithms in their arguments. We’ll consider each algorithm individually, but we’ll fit each to a matrix of attributes (n=2) per player (m=453). We’ll then score not only each player but a grid of values spanning the range of each attribute, to help us visualize the prediction function.

To visualize decision boundaries, we need to take the following steps:

1. Create a 2D meshgrid of input feature values.
2. Apply the decision\_function to each point on the meshgrid, which requires unstacking the grid.
3. Re-shape the predictions back into a grid.
4. Plot the predictions.

We’ll use a 200x200 grid to cover the existing observations plus some padding, but you could adjust the grid to your desired speed and resolution.

import numpy as np  
  
X = df[["OBP","SLG"]].to\_numpy()  
  
GRID\_RESOLUTION = 200  
  
disp\_x\_range, disp\_y\_range = ( (.6\*X[:,i].min(), 1.2\*X[:,i].max())   
 for i in [0,1]  
 )  
xx, yy = np.meshgrid(np.linspace(\*disp\_x\_range, GRID\_RESOLUTION),   
 np.linspace(\*disp\_y\_range, GRID\_RESOLUTION)  
 )  
grid\_shape = xx.shape  
grid\_unstacked = np.c\_[xx.ravel(), yy.ravel()]

**Elliptic Envelope**

The shape of the elliptic envelope is determined by the data’s covariance matrix, which gives the variance of feature i on the main diagonal [i,i] and the covariance of features i and j in the [i,j] positions. Because the covariance matrix is sensitive to outliers, this algorithm uses the Minimum Covariance Determinant (MCD) Estimator, which is [recommended](https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html#sklearn.covariance.MinCovDet) for unimodal and symmetric distributions, with shuffling determined by the random\_state input for reproducibility. This robust covariance matrix will come in handy again later.

Because we want to compare the outlier scores in their ranking rather than a binary outlier/inlier classification, we use the decision\_function to score players.

from sklearn.covariance import EllipticEnvelope  
  
ell = EllipticEnvelope(random\_state=17).fit(X)  
df["outlier\_score\_ell"] = ell.decision\_function(X)  
Z\_ell = ell.decision\_function(grid\_unstacked).reshape(grid\_shape)

**Local Outlier Factor**

This approach to measuring isolation is based on k-nearest neighbors (KNN). We calculate the total distance from each observation to its nearest neighbors to define local density, and then we compare each observation’s local density with that of its neighbors. Observations with local density much less than their neighbors are considered outliers.

**Choosing the number of neighbors to include:** In KNN, a rule of thumb is to let K = sqrt(N), where N is your observation count. From this rule, we obtain a K close to 20 (which happens to be the default K for LOF). You can increase or decrease K to reduce overfitting or underfitting, respectively.

K = int(np.sqrt(X.shape[0]))  
  
print(f"Using K={K} nearest neighbors.")

Using K=19 nearest neighbors.

**Choosing a distance measure:** Note that our features are correlated and have different variances, so Euclidean distance is not very meaningful. We will use Mahalanobis distance, which accounts for feature scale and correlation.

In calculating the Mahalanobis distance, we’ll use the robust covariance matrix. If we had not already calculated it via Ellliptic Envelope, we could calculate it [directly](https://scikit-learn.org/stable/modules/generated/sklearn.covariance.MinCovDet.html).

from scipy.spatial.distance import pdist, squareform  
  
# If we didn't have the elliptical envelope already,  
# we could calculate robust covariance:  
# from sklearn.covariance import MinCovDet  
# robust\_cov = MinCovDet().fit(X).covariance\_  
# But we can just re-use it from elliptical envelope:  
robust\_cov = ell.covariance\_  
  
print(f"Robust covariance matrix:\n{np.round(robust\_cov,5)}\n")  
  
inv\_robust\_cov = np.linalg.inv(robust\_cov)  
  
D\_mahal = squareform(pdist(X, 'mahalanobis', VI=inv\_robust\_cov))  
  
print(f"Mahalanobis distance matrix of size {D\_mahal.shape}, "  
 f"e.g.:\n{np.round(D\_mahal[:5,:5],3)}...\n...\n")

Robust covariance matrix:  
[[0.00077 0.00095]  
 [0.00095 0.00366]]  
  
Mahalanobis distance matrix of size (362, 362), e.g.:  
[[0. 2.86 1.278 0.964 0.331]  
 [2.86 0. 2.63 2.245 2.813]  
 [1.278 2.63 0. 0.561 0.956]  
 [0.964 2.245 0.561 0. 0.723]  
 [0.331 2.813 0.956 0.723 0. ]]...  
...

**Fitting the Local Outlier Factor:** Note that using a custom distance matrix requires us to pass metric="precomputed" to the constructor and then the distance matrix itself to the fit method. (See [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html#sklearn.neighbors.LocalOutlierFactor.fit) for more details.)

Also note that unlike other algorithms, with LOF we are [instructed](https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_novelty_detection.html) not to use the score\_samples method for scoring existing observations; this method should only be used for novelty detection.

from sklearn.neighbors import LocalOutlierFactor  
  
lof = LocalOutlierFactor(n\_neighbors=K, metric="precomputed", novelty=True  
 ).fit(D\_mahal)  
  
df["outlier\_score\_lof"] = lof.negative\_outlier\_factor\_

**Create the decision boundary:** Because we used a custom distance metric, we must also compute that custom distance between each point in the grid to the original observations. Before we used the spatial measure pdist for pairwise distances between each member of a single set, but now we use cdist to return the distances from each member of the first set of inputs to each member of the second set.

from scipy.spatial.distance import cdist  
  
D\_mahal\_grid = cdist(XA=grid\_unstacked, XB=X,   
 metric='mahalanobis', VI=inv\_robust\_cov  
 )  
Z\_lof = lof.decision\_function(D\_mahal\_grid).reshape(grid\_shape)

**Support Vector Machine (SGD-One-Class SVM)**

SVMs use the kernel trick to transform features into a higher dimensionality where a separating hyperplane can be identified. The radial basis function (RBF) kernel requires the inputs to be standardized, but as the [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) for StandardScaler notes, that scaler is sensitive to outliers, so we'll use RobustScaler. We'll pipe the scaled inputs into Nyström kernel approximation, as suggested by the [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDOneClassSVM.html) for SGDOneClassSVM.

from sklearn.pipeline import make\_pipeline  
from sklearn.preprocessing import RobustScaler  
from sklearn.kernel\_approximation import Nystroem  
from sklearn.linear\_model import SGDOneClassSVM  
  
suv = make\_pipeline(  
 RobustScaler(),  
 Nystroem(random\_state=17),  
 SGDOneClassSVM(random\_state=17)  
).fit(X)  
  
df["outlier\_score\_svm"] = suv.decision\_function(X)  
  
Z\_svm = suv.decision\_function(grid\_unstacked).reshape(grid\_shape)

**Isolation Forest**

This tree-based approach to measuring isolation performs random recursive partitioning. If the average number of splits required to isolate a given observation is *low*, that observation is considered a *stronger* candidate outlier. Like Random Forests and other tree-based models, Isolation Forest does not assume that the features are normally distributed or require them to be scaled. By default, it builds 100 trees. Our example only uses two features, so we do not enable feature sampling.

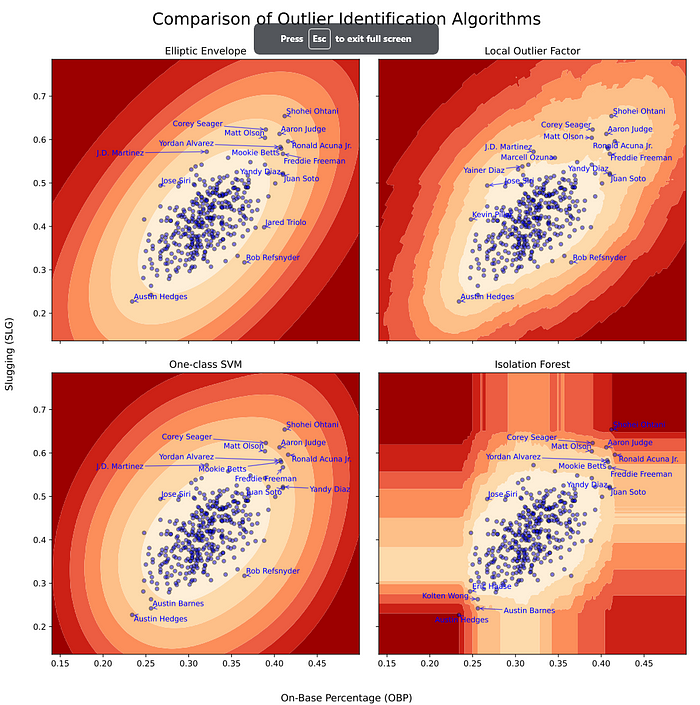
from sklearn.ensemble import IsolationForest  
  
iso = IsolationForest(random\_state=17).fit(X)  
  
df["outlier\_score\_iso"] = iso.score\_samples(X)  
  
Z\_iso = iso.decision\_function(grid\_unstacked).reshape(grid\_shape)

**Results: inspecting decision boundaries**

Note that the predictions from these models have different distributions. We apply QuantileTransformer to make them more visually comparable on a given grid. From the [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.QuantileTransformer.html), please note:

Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

from adjustText import adjust\_text  
from sklearn.preprocessing import QuantileTransformer  
  
N\_QUANTILES = 8 # This many color breaks per chart  
N\_CALLOUTS=15 # Label this many top outliers per chart  
  
fig, axs = plt.subplots(2, 2, figsize=(12, 12), sharex=True, sharey=True)  
  
fig.suptitle("Comparison of Outlier Identification Algorithms",size=20)  
fig.supxlabel("On-Base Percentage (OBP)")  
fig.supylabel("Slugging (SLG)")  
  
ax\_ell = axs[0,0]  
ax\_lof = axs[0,1]  
ax\_svm = axs[1,0]  
ax\_iso = axs[1,1]  
  
model\_abbrs = ["ell","iso","lof","svm"]  
  
qt = QuantileTransformer(n\_quantiles=N\_QUANTILES)  
  
for ax, nm, abbr, zz in zip( [ax\_ell,ax\_iso,ax\_lof,ax\_svm],   
 ["Elliptic Envelope","Isolation Forest",  
 "Local Outlier Factor","One-class SVM"],   
 model\_abbrs,  
 [Z\_ell,Z\_iso,Z\_lof,Z\_svm]  
 ):  
 ax.title.set\_text(nm)  
 outlier\_score\_var\_nm = f"outlier\_score\_{abbr}"  
   
 qt.fit(np.sort(zz.reshape(-1,1)))  
 zz\_qtl = qt.transform(zz.reshape(-1,1)).reshape(zz.shape)  
  
 cs = ax.contourf(xx, yy, zz\_qtl, cmap=plt.cm.OrRd.reversed(),   
 levels=np.linspace(0,1,N\_QUANTILES)  
 )  
 ax.scatter(X[:, 0], X[:, 1], s=20, c="b", edgecolor="k", alpha=0.5)  
   
 df\_callouts = df.sort\_values(outlier\_score\_var\_nm).head(N\_CALLOUTS)  
 texts = [ ax.text(row["OBP"], row["SLG"], row["Name"], c="b",  
 size=9, alpha=1.0)   
 for \_,row in df\_callouts.iterrows()  
 ]  
 adjust\_text(texts,   
 df\_callouts["OBP"].values, df\_callouts["SLG"].values,   
 arrowprops=dict(arrowstyle='->', color="b", alpha=0.6),   
 ax=ax  
 )  
  
plt.tight\_layout(pad=2)  
plt.show()  
  
for var in ["OBP","SLG"]:  
 df[f"Pctl\_{var}"] = 100\*(df[var].rank()/df[var].size).round(3)  
  
model\_score\_vars = [f"outlier\_score\_{nm}" for nm in model\_abbrs]   
model\_rank\_vars = [f"Rank\_{nm.upper()}" for nm in model\_abbrs]  
  
  
df[model\_rank\_vars] = df[model\_score\_vars].rank(axis=0).astype(int)  
   
# Averaging the ranks is arbitrary; we just need a countdown order  
df["Rank\_avg"] = df[model\_rank\_vars].mean(axis=1)  
  
print("Counting down to the greatest outlier...\n")  
print(  
 df.sort\_values("Rank\_avg",ascending=False  
 ).tail(N\_CALLOUTS)[["Name","AB","PA","H","2B","3B",  
 "HR","BB","HBP","SO","OBP",  
 "Pctl\_OBP","SLG","Pctl\_SLG"  
 ] +   
 [f"Rank\_{nm.upper()}" for nm in model\_abbrs]  
 ].to\_string(index=False)  
)



Counting down to the greatest outlier...  
  
 Name AB PA H 2B 3B HR BB HBP SO OBP Pctl\_OBP SLG Pctl\_SLG Rank\_ELL Rank\_ISO Rank\_LOF Rank\_SVM  
 Austin Barnes 178 200 32 5 0 2 17 2 43 0.256 2.6 0.242 0.6 19 7 25 12  
 J.D. Martinez 432 479 117 27 2 33 34 2 149 0.321 52.8 0.572 98.1 15 18 5 15  
 Yandy Diaz 525 600 173 35 0 22 65 8 94 0.410 99.2 0.522 95.4 13 15 13 10  
 Jose Siri 338 364 75 13 2 25 20 2 130 0.267 5.5 0.494 88.4 8 14 15 13  
 Juan Soto 568 708 156 32 1 35 132 2 129 0.410 99.2 0.519 95.0 12 13 11 11  
 Mookie Betts 584 693 179 40 1 39 96 8 107 0.408 98.6 0.579 98.3 7 10 20 7  
 Rob Refsnyder 202 243 50 9 1 1 33 5 47 0.365 90.5 0.317 6.6 5 19 2 14  
 Yordan Alvarez 410 496 120 24 1 31 69 13 92 0.407 98.3 0.583 98.6 6 9 18 6  
 Freddie Freeman 637 730 211 59 2 29 72 16 121 0.410 99.2 0.567 97.8 9 11 9 8  
 Matt Olson 608 720 172 27 3 54 104 4 167 0.389 96.5 0.604 99.2 11 6 7 9  
 Austin Hedges 185 212 34 5 0 1 11 2 47 0.234 0.3 0.227 0.3 10 1 4 3  
 Aaron Judge 367 458 98 16 0 37 88 0 130 0.406 98.1 0.613 99.4 3 5 6 4  
Ronald Acuna Jr. 643 735 217 35 4 41 80 9 84 0.416 100.0 0.596 98.9 2 3 10 2  
 Corey Seager 477 536 156 42 0 33 49 4 88 0.390 97.0 0.623 99.7 4 4 3 5  
 Shohei Ohtani 497 599 151 26 8 44 91 3 143 0.412 99.7 0.654 100.0 1 2 1 1

**Analysis and Conclusions**

It looks like the four implementations mostly agree on how to define outliers, but with some noticeable differences in scores and also in ease of use.

**Elliptic Envelope** has narrower contours around the ellipse’s minor axis, so it tends to highlight those interesting players who run contrary to the overall correlation between features. For example, Rays outfielder José Siri ranks as more of an outlier under this algorithm due to his high SLG (88th percentile) versus low OBP (5th percentile), which is consistent with an aggressive hitter who swings hard at borderline pitches and either crushes them or gets weak-to-no contact.

Elliptic Envelope is also easy to use without configuration, and it provides the robust covariance matrix. If you have low-dimensional data and a reasonable expectation for it to be normally distributed (which is often not the case), you might want to try this simple approach first.

**One-class SVM** has more uniformly spaced contours, so it tends to emphasize observations along the overall direction of correlation more than the Elliptic Envelope. All-Star first basemen Freddie Freeman (Dodgers) and Yandy Diaz (Rays) rank more strongly under this algorithm than under others, since their SLG and OBP are both excellent (99th and 97th percentile for Freeman, 99th and 95th for Diaz).

The RBF kernel required an extra step for standardization, but it also seemed to work well on this simple example without fine-tuning.

**Local Outlier Factor** picked up on the “cluster of excellence” mentioned earlier with a small bimodal contour (barely visible in the chart). Since the Dodgers’ outfielder/second-baseman Mookie Betts is surrounded by other excellent hitters including Freeman, Yordan Alvarez, and Ronald Acuña Jr., he ranks as only the 20th-strongest outlier under LOF, versus 10th or stronger under the other algorithms. Conversely, Braves outfielder Marcell Ozuna had slightly lower SLG and considerably lower OBP than Betts, but he is more of an outlier under LOF because his neighborhood is less dense.

LOF was the most time-consuming to implement since we created robust distance matrices for fitting and scoring. We could have spent some time tuning K as well.

**Isolation Forest**tends to emphasize observations at the corners of the feature space, because splits are distributed across features. Backup catcher Austin Hedges, who played for the Pirates and Rangers in 2023 and signed with Guardians for 2024, is strong defensively but the worst batter (with at least 200 plate appearances) in both SLG and OBP. Hedges can be isolated in a single split on either OBP or OPS, making him the strongest outlier. Isolation Forest is the *only*algorithm that didn’t rank Shohei Ohtani as the strongest outlier: since Ohtani was edged out in OBP by Ronald Acuña Jr., both Ohtani and Acuña can be isolated in a single split on only *one*feature.

As with common *supervised*tree-based learners, Isolation Forest does not extrapolate, making it better suited for fitting to a contaminated dataset for outlier detection than for fitting to an anomaly-free dataset for novelty detection (where it wouldn’t score new outliers more strongly than the existing observations).

Although Isolation Forest worked well out of the box, its failure to rank **Shohei Ohtani** as the *greatest outlier in baseball (and probably all professional sports)* illustrates the primary limitation of any outlier detector: the data you use to fit it.

Not only did we omit defensive stats (sorry, Austin Hedges), we didn’t bother to include *pitching* stats. Because pitchers don’t even try to hit anymore… except for Ohtani, whose season included the second-best batting average against (BAA) and 11th-best earned run average (ERA) in baseball (minimum 100 innings), a complete-game shutout, and a game in which he struck out ten batters and hit two home runs.

It has been suggested that Shohei Ohtani is an advanced extraterrestrial impersonating a human, but it seems more likely that there are **two**advanced extraterrestrials impersonating the same human. Unfortunately, one of them just had elbow surgery and won’t pitch in 2024… but the other just signed a record 10-year, $700 million contract. And thanks to outlier detection, now we can see why!