Solution - Foot-Moneyball

Solution: Foot-Moneyball

Import Libraries and Dataset

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('whitegrid')
```

In [2]:
```
df = pd.read_csv('data/foot-moneyball.csv')
```

In [3]:
```
qbs = pd.read_csv('data/quarterbacks.csv')
```

Exploratory Analysis and Data Cleaning

For this challenge, we'll include an abbreviated section for exploratory analysis and data cleaning, since we know the datasets have been pre-cleaned.

There are still a few key things we should check to familiarize ourselves with the dataset, starting with example observations.

In [4]:
```
# Example observations
df.head()
```

Out[4]:

	season	date	name	team	team_score	opponent	opponent_score	location	passing_attempts	com
0	2000	2000-09-10	Trent Dilfer	BAL	39	JAX	36	H	0	0
1	2000	2000-10-01	Trent Dilfer	BAL	12	CLE	0	A	0	0
2	2000	2000-10-22	Trent Dilfer	BAL	6	TEN	14	H	13	7
3	2000	2000-10-29	Trent Dilfer	BAL	6	PIT	9	H	25	12
4	2000	2000-11-05	Trent Dilfer	BAL	27	CIN	7	A	34	23

In [5]:
```
qbs.head()
```

Out[5]:

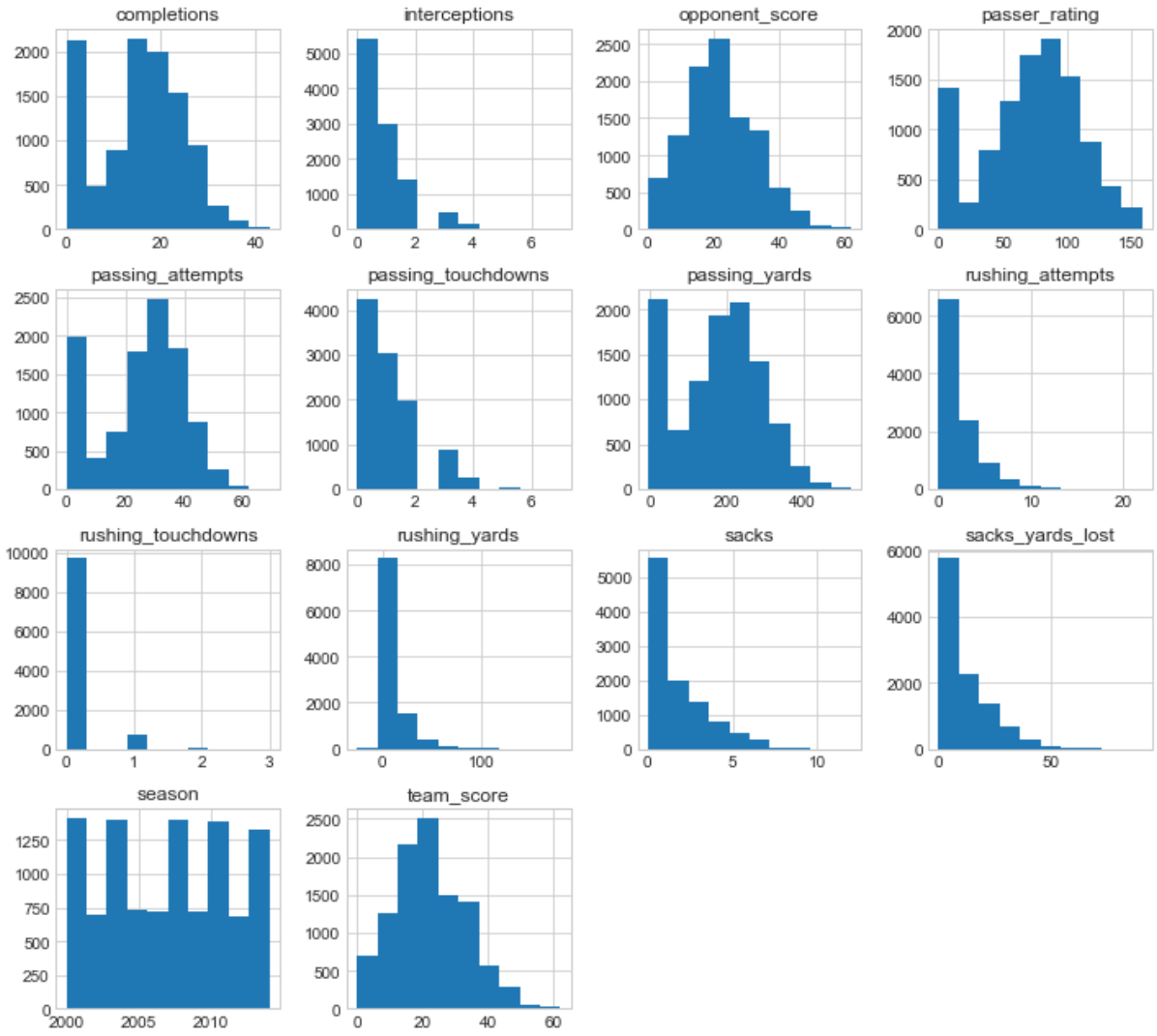
	name	draft_round	college
0	Trent Dilfer	1.0	Fresno St.
1	Jeff Lewis	4.0	Northern Arizona
2	Kordell Stewart	2.0	Colorado
3	Connor Shaw	NaN	South Carolina
4	Anthony Wright	NaN	South Carolina

Two important insights right off the bat:

1. The games dataset includes games in which a quarterback did not play (or at least did not record any stats). We'll want to keep this in mind as we continue to analysis.
2. Some quarterbacks having missing values for `draft_round`. Based on the challenge description, we know that these are undrafted quarterbacks.

Next, let's look at distributions of key features.

In [6]:
```
# Numeric distributions
df.hist(figsize=(12,11))
plt.show()
```



These are look reasonable. Let's also look at the distribution of draft rounds for quarterbacks.

In [7]:
```
# Percent undrafted
print('{} of {} were undrafted.'.format(qbs.draft_round.isnull().sum(),
                                        len(qbs)))
print('Those drafted were in rounds:')

# Numeric distributions
qbs[qbs.draft_round.notnull()].hist(figsize=(4,3))
plt.show()
```

```
53 of 260 were undrafted.
Those drafted were in rounds:
```

This also makes sense. Since the quarterback position is such a high priority position, we'd expect most of those drafted to be in earlier rounds.

Also note that while there are currently 7 rounds in the NFL draft, there used to be more rounds.

In [8]:
```
qbs[qbs.draft_round == qbs.draft_round.max()]
```

Out[8]:

	name	draft_round	college
47	Doug Flutie	11.0	Boston Col.

Finally, let's check for missing values, just in case.

In [9]:
```
# Check for missing values
df.isnull().sum()
```

Out[9]:
```
season              0
date                0
name                0
team                0
team_score          0
opponent            0
opponent_score      0
location            0
passing_attempts    0
completions         0
interceptions       0
passer_rating       0
sacks               0
sacks_yards_lost    0
passing_touchdowns  0
passing_yards       0
rushing_attempts    0
rushing_touchdowns  0
rushing_yards       0
dtype: int64
```

Create a model for passer rating

Our first objective is to create a machine learning model for passer rating. Even before training this model, we can anticipate that the model predictions **will most likely not** exactly match the calculated numbers.

The reason is that most machine learning models are designed to be *generalizable.* And because (1) the algorithm does not know that the values were calculated deterministically and (2) the dataset size is relatively small, we would expect it to still include some element of uncertainty.

In [10]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_absolute_error
```

In this case, to minimize noise, we should first hand-pick the features based on what's actually used in the original calculation.

In [11]:
```
abt = df[['passing_attempts', 'completions', 'passing_touchdowns',
              'interceptions', 'passing_yards', 'passer_rating']]
```

Next, we'll train a baseline random forest model.

In [12]:
```
# Split input features and target variable
y = abt.passer_rating
X = abt.drop('passer_rating', axis=1)

# Split training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1

# Fit baseline RF moel
rf = RandomForestRegressor(random_state=101)
rf.fit(X_train, y_train)

# Predict passer rating on test set
pred = rf.predict(X_test)
```

To evaluate our model, we can look at 4 outputs.

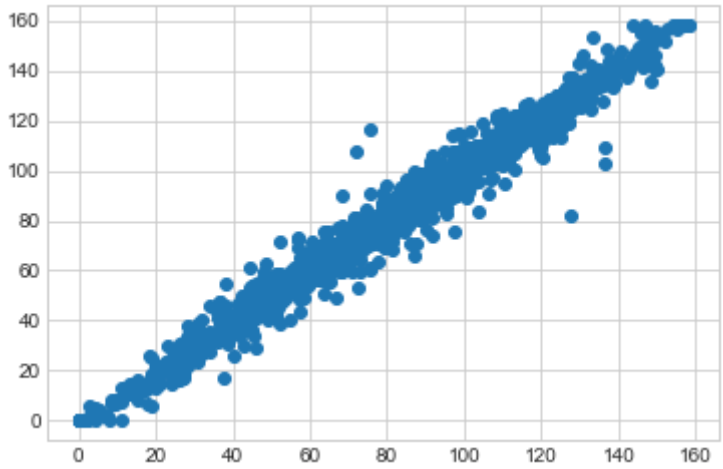
1. Model R-squared
2. Mean absolute error (on average, how many points were we off)
3. Percent correct (actual percent that match) - For this, to account for rounding, we'll simply check if our predicted value is within 0.1 of the actual.
4. The plot of predicted vs. actual passer ratings

In [13]:
```
def percent_correct(actual, predicted):
    corrects = [abs(p - a) <= 0.1 for a,p in zip(actual, predicted)]
    return round(np.mean(corrects) * 100,1)

# Model
```

```
print( 'R^2:', r2_score(y_test, pred))
print( 'MAE:', mean_absolute_error(y_test, pred))
print( 'Pct Correct:', percent_correct(y_test, pred) )

# Plot predicted vs actual
plt.scatter(pred, y_test)
```

```
R^2: 0.987617750242182
MAE: 2.6272536287242176
Pct Correct: 18.4
```

Out[13]: <matplotlib.collections.PathCollection at 0x10f3dac18>



Ass you can see, our model is quite strong: we have almost 0.99 R-squared, and we're off by 3 points on average. Even so, it's difficult for our model to perfectly re-create the calculated passer ratings (unless the model is purposefully tuned to "overfit").

Calculating passer rating

Now let's compare our results if we calculate passer rating directly.

First, we'll write the function based on the formula.

In [14]:
```
def calc_passer_rating(X):
    # Calculate variables
    a = (X.completions / X.passing_attempts - 0.3)*5.
    b = (X.passing_yards / X.passing_attempts - 3)*0.25
    c = (X.passing_touchdowns / X.passing_attempts)*20.
    d = 2.375 - (X.interceptions / X.passing_attempts * 25.)

    # Handle divide-by-zero cases where passing_attemptions == 0
    a.fillna(0, inplace=True)
    b.fillna(0, inplace=True)
    c.fillna(0, inplace=True)
    d.fillna(0, inplace=True)
```

```
    # Cap at 0 and 2.375
    a.clip(0, 2.375, inplace=True)
    b.clip(0, 2.375, inplace=True)
    c.clip(0, 2.375, inplace=True)
    d.clip(0, 2.375, inplace=True)

    # Calculated passer rating
    calc = ((a + b + c + d)/6.)*100.

    return calc
```

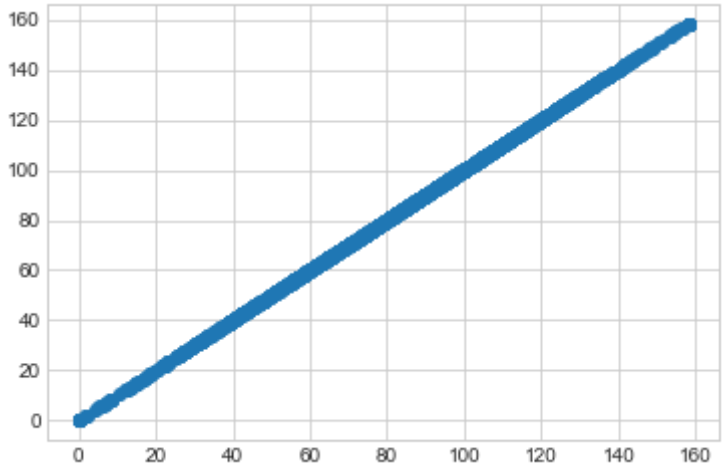
We'll perform the calculations, then run the same evaluation metrics as above.

In [15]:
```
calc = calc_passer_rating(X_test)
```

In [16]:
```
# Calculations
print( 'R^2:', r2_score(y_test, calc))
print( 'MAE:', mean_absolute_error(y_test, calc))
print( 'Pct Correct:', percent_correct(y_test, calc) )

# Plot calculated vs actual
plt.scatter(calc, y_test)
```

```
R^2: 0.9999995159192752
MAE: 0.02202836074984922
Pct Correct: 100.0
```

Out[16]:
```
<matplotlib.collections.PathCollection at 0x10f2f5198>
```



As you can see, the direct calculation here is still more precise. (The reason the percent correct is 100% while MAE > 0.0 is due to rounding differences.)

Tip: An exceptional candidate will take this opportunity to explain the lesson learned from this exercise. There are two important lessons here.

- The first is that machine learning does not replace domain knowledge. As a data scientist, it's not enough to just "throw features into an algorithm." That might get you decent results, but not the best results. Data scientists should spend just as much time learning about their industries.
- The second lesson is that data scientists must also understand when machine learning is *not* the best solution vs. traditional methods.

Feature Engineering

Next, before we create groupings of quarterbacks based on their passing vs. rushing ability, we should engineer some new features. We don't want to run the clusters on the base features directly (or even run PCA) because not all of the features have equal (actual) importance. **Instead, for each quarterback, we will calculate and aggregate passer rating and "rusher rating" across all games.**

First, we'll start by creating an indicator variable for whether the quarterback actually played in a game.

In [17]:
```
# Check if the quarterback passed, rushed, or got sacked at least once
df['played'] = ((df.passing_attempts + df.sacks + df.rushing_attempts) >
```

Then, we'll aggregate stats across all their games played in the dataset.

In [18]:
```
# Aggregate for each quarterback
gb = df.groupby('name')

# Relevant passing and rushing stats
agg_stats = gb[['played', 'passing_attempts', 'completions', 'interceptic
                'passing_touchdowns', 'passing_yards',
                'rushing_attempts', 'rushing_touchdowns', 'rushing_yards'

# Only include quarterbacks who have played at least 50 games
agg_stats = agg_stats[agg_stats.played >= 50]
```

In [19]:
```
agg_stats.head()
```

Out[19]:

	played	passing_attempts	completions	interceptions	passing_touchdowns	passing_yards	ru
name							
Aaron Brooks	95	3040	1719	95	129	20822	37
Aaron Rodgers	121	3862	2539	64	249	31561	41
Alex Smith	113	3263	1967	76	131	21731	35
Andrew Luck	54	2073	1209	55	95	14786	20
Andy Dalton	68	2269	1389	72	100	15631	22

We can calculate an aggregate passer rating for each quarterback using our function from earlier.

In [20]:
```
agg_stats['passer_rating'] = calc_passer_rating(agg_stats)
```

For the "rusher rating," we're going to mimic the passer rating, with a few small tweaks.

First, we're going to double the multipliers to reflect that it's generally tougher to get rushing yards as a quarterback (by the way, it doesn't actually matter if you change the multipliers or not as long as you use the same calculation for every quarterback; we are not comparing passer rating directly with rusher rating, but rather the *relative* ratings between quarterbacks).

Second, we're going to halve the denominator of the last calculation to reflect the fact that we're only include 2 variables (rushing yards and touchdowns) instead of 4 (completions, interceptions, touchdowns, and yards).

Tip: For this type of challenge, *especially if you don't have too much domain knowledge*, it's not the output that's most important but rather your reasoning and logic! As long as you pick a reasonable solution and provide a clear explanation for it, then that's more than good enough.

In [21]:
```
def calc_rusher_rating(X):
    # Calculate variables
    a = (X.rushing_yards / X.rushing_attempts - 3)*0.5
    b = (X.rushing_touchdowns / X.rushing_attempts)*40.

    # Handle divide-by-zero cases where rushing_attempts == 0
    a.fillna(0, inplace=True)
    b.fillna(0, inplace=True)

    # Cap at 0 and 2.375
```

```
    a.clip(0, 2.375, inplace=True)
    b.clip(0, 2.375, inplace=True)

    # Calculated passer rating
    calc = ((a + b)/3.)*100.

    return calc
```

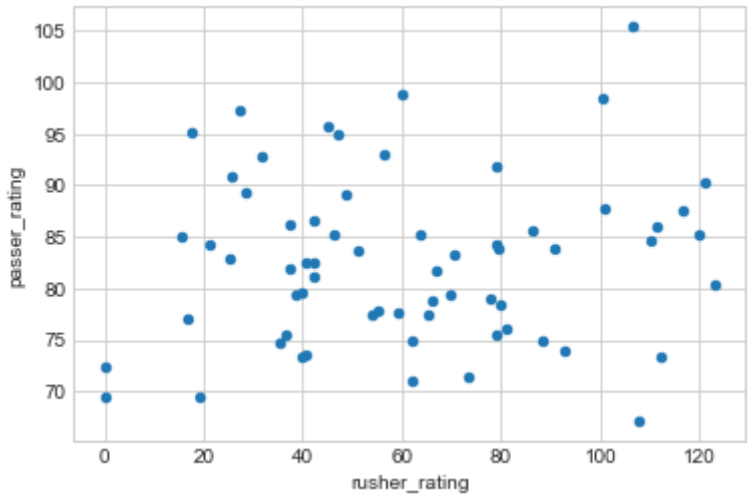
In [22]:
```
agg_stats['rusher_rating'] = calc_rusher_rating(agg_stats)
```

Finally, we'll create a new analytical base table for clustering. It will only have the aggregate passer rating and rusher rating features.

In [23]:
```
abt_cl = agg_stats.loc[:,['passer_rating', 'rusher_rating']]
```

In [24]:
```
abt_cl.plot(x='rusher_rating', y='passer_rating', kind='scatter')
```

Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x10f281358>



Passers vs. Rushers (Clustering)

Now we're ready to create our groupings/clusters. We will use the simple, popular K-Means algorithm. It's one of the most versatile clustering algorithms, and the key to successful cluster lies

in the feature engineering *before* running the algorithm.

In [25]:
```
from sklearn.cluster import KMeans
```

One note with clustering is that if you're using a distance-based algorithm, it's important to standardize or normalize your features if they are on different scales.

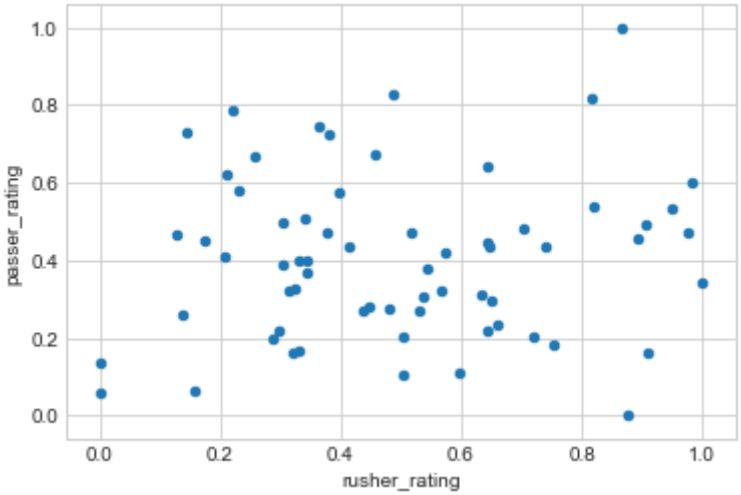
In our case, it actually looks like passer rating and rusher rating are on comparable scales. Nonetheless, it doesn't hurt to normalize our features first.

In [26]:
```
# Important to do if your engineered features are on different scales
abt_norm = (abt_cl - abt_cl.min())/(abt_cl.max() - abt_cl.min())
```

This scales our feature values to be between 0 and 1.

In [27]:
```
abt_norm.plot(x='rusher_rating', y='passer_rating', kind='scatter')
```

Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10f26add8>
```



Let's create our clusters.

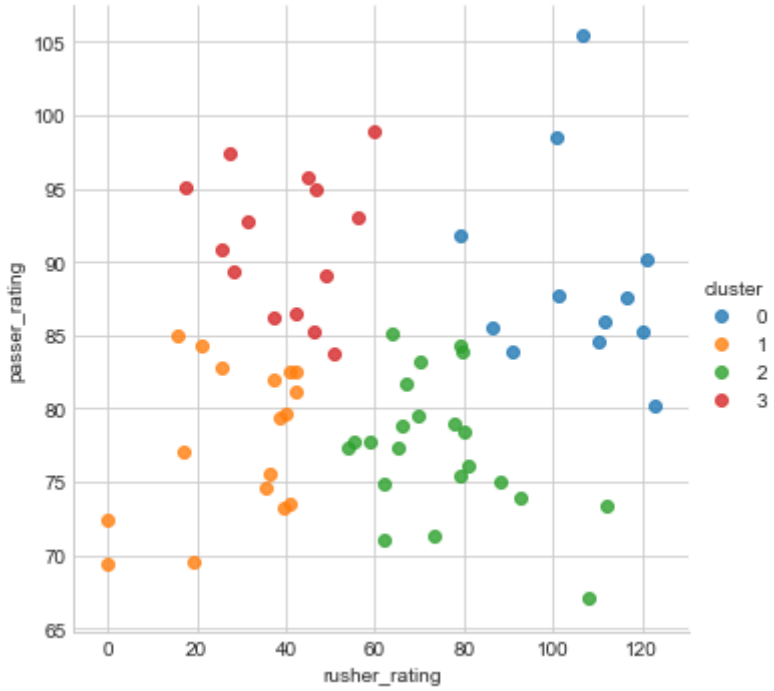
We will create 4 clusters that (hopefully) correspond with the 4 quadrants in our performance chart.

In [28]:
```

```

```
kmeans = KMeans(n_clusters=4, random_state=123)
kmeans.fit(abt_norm)

abt_cl['cluster'] = kmeans.predict(abt_norm)
```

In [29]:
```
sns.lmplot(x='rusher_rating',
           y='passer_rating',
           hue='cluster',
           fit_reg=False,
           data=abt_cl)
```

Out[29]: <seaborn.axisgrid.FacetGrid at 0x10f268c18>



Just for fun, let's give our clusters some new names.

In [30]:
```
abt_cl.cluster.replace({0:'Versatile',
                        1:'Upgradeable',
                        2:'Track Star',
                        3:'Gunslinger'}, inplace=True)
```

In [31]:
```
versatile_qbs = abt_cl[abt_cl.cluster == 'Versatile'].reset_index()
```

Moneyball Value Pick

There are many ways to make your "moneyball value pick." Your moneyball pick may be completely different from ours depending your analysis path, your final clusters, and even your criteria for a good pick. The key is not who you pick, but rather your reasoning behind the pick.

For our analysis, our pick will be the player in the "Versatile" group who was drafted in the latest

In [32]:
```
versatile_qbs.merge(qbs, on='name').sort_values(by='draft_round')
```

Out[32]:

	name	passer_rating	rusher_rating	cluster	draft_round	college
0	Aaron Rodgers	105.506646	106.650446	Versatile	1.0	California
1	Andrew Luck	84.624739	110.047847	Versatile	1.0	Stanford
2	Cam Newton	85.203786	119.974747	Versatile	1.0	Auburn
4	Daunte Culpepper	87.558131	116.721802	Versatile	1.0	Central Florida
6	Donovan McNabb	85.983362	111.414457	Versatile	1.0	Syracuse
8	Michael Vick	80.265290	123.052434	Versatile	1.0	Virginia Tech
11	Steve McNair	83.865100	90.920398	Versatile	1.0	Alcorn St.
3	Colin Kaepernick	90.202306	121.100427	Versatile	2.0	Nevada
10	Russell Wilson	98.455410	100.617284	Versatile	3.0	Wisconsin
5	David Garrard	85.574940	86.410256	Versatile	4.0	East Carolina
9	Rich Gannon	91.824833	79.201681	Versatile	4.0	Delaware
7	Jeff Garcia	87.726218	101.058201	Versatile	NaN	San Jose St.

The most versatile value pick wasn't even drafted at all. Jeff Garcia was an undrafted free agent who later went on to become a 4x pro-bowler in the NFL.