Modern Methods of Data Analysis

Professor: Boris Mirkin

Home Assignment

Team "Moneyball"
Alexander Borzunov
Arthur Liss
Roman Misiutin

Moscow, 2018

Contents

1 Data Description

In this report, we use so-called Moneyball data on shots made during the NBA season of 2014-2015. The initial dataset consists of 122502 shot records made during 896 games of the season. Each shot described by 18 features:

1. `GAME_ID` – numerical identifier of the game
2. `MATCHUP` – date and participating teams
3. `LOCATION` – whether the game is taking place at the home stadium of the player or not
4. `SHOT_NUMBER` – number of shots attempted by the team
5. `PERIOD` – period of the game
6. `GAME_CLOCK` – time until the end of the period
7. `SHOT_CLOCK` – time until the shot clock violation
8. `DRIBBLES`– number of dribbles made by the player with the ball
9. `TOUCH_TIME` – how long the shooting player has possession of the ball
10. `SHOT_DIST` – the distance between the shooting player and the basket before making the shot
11. `PTS_TYPE` – type of the attempted shot
12. `SHOT_RESULT` – whether the shot was successful or not
13. `CLOSEST_DEFENDER` – name of the closest player from opposing team
14. `CLOSEST_DEFENDER_ID` – his identifier
15. `CLOSE_DEF_DIST` – distance between the shooting player and the defender
16. `PTS` – points scored by the shot
17. `player_name` – name of the shooting player
18. `player_id` – identifier of the shooting player

To comply with requirements imposed on the size of a dataset, we consider only shots made in one game. Arbitrarily, we choose the game on October 28, 2014, between New Orleans Pelicans and Orlando Magic (`GAME_ID=21400001`).

Now we can omit `GAME_ID` and `MATCHUP`. Also, we exclude from further analysis features that trivially depend on other features (`FGM`, `PTS`, `player_name`, `CLOSEST_DEFENDER`).

We also need to convert `GAME_CLOCK` to seconds and binary features `LOCATION`, `SHOT_RESULT` to its numerical representation.

```
In [4]: match_data = data[data.GAME_ID == 21400001]
        match_data.index = np.arange(match_data.shape[0])

        features_to_drop = ['GAME_ID',
                            'MATCHUP',
                            'PTS',
                            'player_name',
                            'CLOSEST_DEFENDER',
                           ]
        match_data.drop(features_to_drop, axis=1, inplace=True)

        match_data['SHOT_RESULT'] = match_data['SHOT_RESULT'].map({'made':1,'missed':0})
        match_data['LOCATION'] = match_data['LOCATION'].map({'H':1,'A':0})

        def ms2s(clock):
            m,s = map(int,clock.split(':'))
            return m*60 + s

        match_data['GAME_CLOCK'] = match_data['GAME_CLOCK'].map(lambda x: ms2s(x))

In [5]: print(match_data.shape)
        match_data.head().T
```

```
(157, 13)
```

Out[5]:

	0	1	2	3	4
LOCATION	1.0	1.0	1.0	1.0	1.0
SHOT_NUMBER	1.0	2.0	3.0	4.0	5.0
PERIOD	1.0	1.0	1.0	1.0	1.0
GAME_CLOCK	607.0	579.0	435.0	314.0	112.0
SHOT_CLOCK	11.3	13.0	11.5	21.6	15.9
DRIBBLES	0.0	1.0	0.0	0.0	0.0
TOUCH_TIME	0.8	2.9	0.6	0.6	0.8
SHOT_DIST	3.6	0.9	0.8	3.4	2.6
PTS_TYPE	2.0	2.0	2.0	2.0	2.0
SHOT_RESULT	1.0	0.0	1.0	0.0	1.0
CLOSEST_DEFENDER_PLAYER_ID	202696.0	203124.0	202699.0	203095.0	203473.0
CLOSE_DEF_DIST	1.7	2.0	2.6	1.5	4.5
player_id	201600.0	201600.0	201600.0	201600.0	201600.0

Now we have the dataset of 157 records with 13 distinct features.

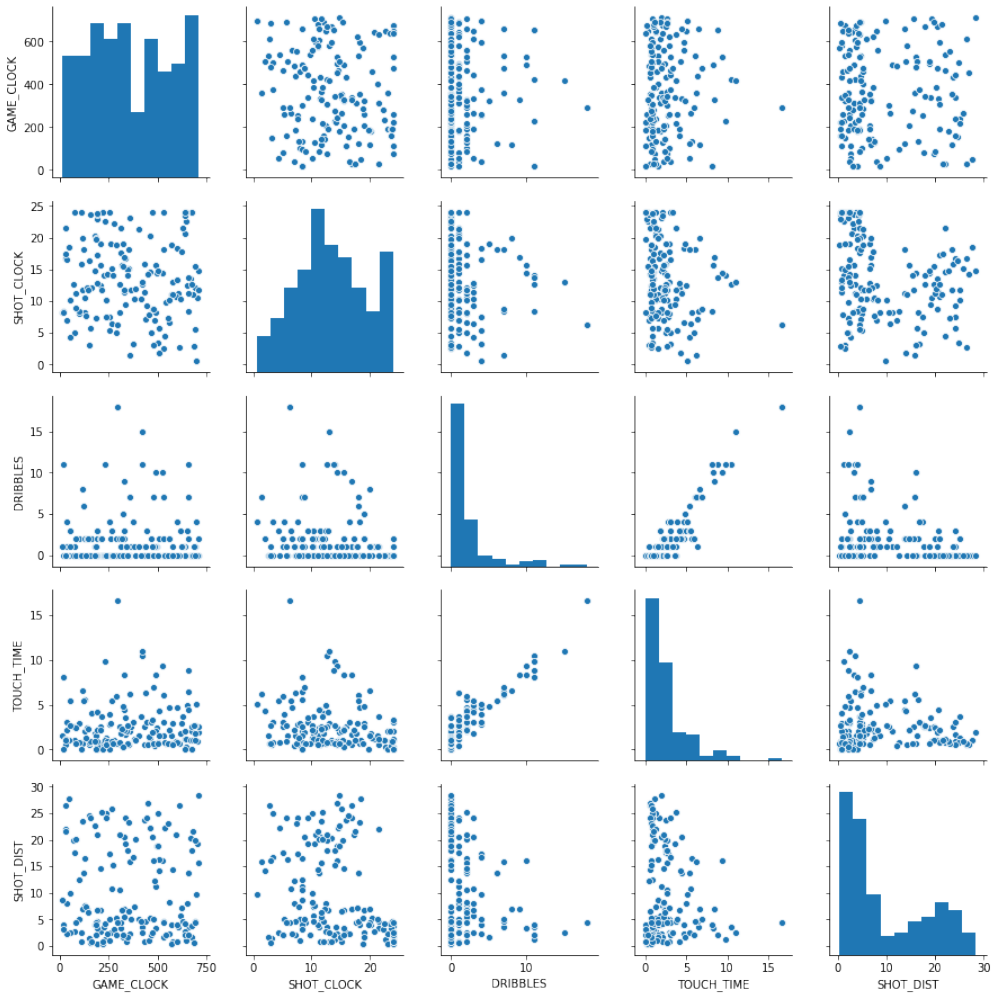
2 Clustering

2.1 Feature Selection

We should choose 3-6 features for the clustering task. To do this, let's consider only numerical features and plot their histograms and pairwise scatterplots.

```
In [6]: numerical_features = ['GAME_CLOCK',
                              'SHOT_CLOCK',
                              'DRIBBLES',
                              'TOUCH_TIME',
                              'SHOT_DIST',
                             ]
        sns.pairplot(match_data[numerical_features])
```

```
Out[6]: <seaborn.axisgrid.PairGrid at 0x7f13344b0908>
```



We see that TOUCH_TIME and DRIBBLES are almost linearly dependent, which is unsurprising given their meaning. Also, GAME_CLOCK is almost uniformly distributed and does not display any obvious interactions with other features. Because of that, we exclude DRIBBLES and GAME_CLOCK features from consideration in this section. As a result, we use SHOT_CLOCK, TOUCH_TIME, SHOT_DIST, CLOSE_DEF_DIST features for clustering. Also, we normalize the data by its range.

```
In [7]: kmeans_features = ['SHOT_CLOCK',
                           'TOUCH_TIME',
                           'SHOT_DIST',
                           'CLOSE_DEF_DIST',
                          ]
        kmeans_data = match_data[kmeans_features]
        data_range = (kmeans_data.max()-kmeans_data.min())
        kmeans_data -= kmeans_data.min()
        kmeans_data /= data_range
```

2.2 KMeans Application

To perform KMeans clustering we use KMeans class form scikit-learn python library. It takes the following parameters:

1. n_clusters – a number of clusters
2. init – a type of initialization:
   - 'random' for randomized initialization
   - 'k-means++' for smart heuristic initialization
   - array for predefined centers
3. n_init – a number of runs to perform. The resulted partition will be optimal with respect to the inertia (or KMeans) criterion:

$$I = \sum_{i=1}^{n} \min_{c_j} d(\bar{x}_i, \bar{c}_j) = \sum_{k=1}^{K} \sum_{i \in S_k} d(\bar{x}_i, \bar{c}_k),$$

   where n – number of points, $\bar{x}_i$ – i-th point. $\bar{c}_j$ – center of cluster j, K – number of clusters, S_k – set of points in cluster k

```
In [8]: km5 = KMeans(n_clusters=5, init='random',n_init=10)
        km5.fit(kmeans_data)
```

```
km9= KMeans(n_clusters=9, init='random',n_init=10)
km9.fit(kmeans_data)

print(f'5-cluster KMeans criterion: {km5.inertia_}')
print(f'9-cluster KMeans criterion: {km9.inertia_}')
```

```
5-cluster KMeans criterion: 9.18399062914476
9-cluster KMeans criterion: 6.1545755593099205
```

2.3 Cluster Interpretation

To interpret obtained partitions we augment the used dataset with the binary feature SHOT_RESULT, so we can explore an influence of partition on shot's success.

```
In [9]: int_features = kmeans_features+ ['SHOT_RESULT']
        int_data = match_data[int_features]
```

We define an auxiliary function for computing cluster-conditional means of given features and their corresponding absolute and relative differences to their grand mean. It takes a data table and a list of cluster labels as parameters.

```
In [10]: def describe_cluster(data,labels):
             X = data.copy()
             X['label'] = labels
             means = X.groupby('label').mean()
             grand_mean = data.mean()
             grand_mean.name = 'grand'
             means = means.append(grand_mean)
             abs_diffs = means.iloc[:-1] - means.iloc[-1]
             rel_diffs = (means.iloc[:-1]/means.iloc[-1] - 1)*100

             smaller = rel_diffs <= -35
             much_smaller = rel_diffs <= -50
             bigger = rel_diffs >= 35
             much_bigger = rel_diffs >= 50

             ans = abs_diffs.copy()
             ans[:] = None
             ans[smaller] = 'smaller'
             ans[much_smaller] = 'much smaller'
             ans[bigger] = 'bigger'
             ans[much_bigger] = 'much bigger'

             return means, abs_diffs,rel_diffs,ans
```

Now we apply this function to the previously obtained partitions

5-cluster partition

```
In [11]: means, a_diff, r_diff, mask = describe_cluster(int_data,km5.labels_)
         print(means,'\n')
         print(a_diff,'\n')
         print(r_diff,'\n')
         print(mask,'\n')
```

gives us

Cluster Mean

Features / Clusters	SHOT_CLOCK	TOUCH_TIME	SHOT_DIST	CLOSE_DEF_DIST	SHOT_RESULT
0	12.018182	9.481818	4.836364	2.454545	0.363636
1	7.226923	2.215385	18.957692	5.107692	0.269231
2	20.735556	1.671111	3.240000	2.911111	0.555556
3	8.991111	2.482222	4.848889	2.691111	0.466667
4	14.913333	1.550000	21.676667	6.196667	0.200000
Grand	13.408917	2.517834	9.938854	3.807643	0.401274

Absolute Difference

Features / Clusters	SHOT'CLOCK	TOUCH'TIME	SHOT'DIST	CLOSE'DEF'DIST	SHOT'RESULT
0	-1.390735	6.963984	-5.102490	-1.353098	-0.037638
1	-6.181994	-0.302450	9.018839	1.300049	-0.132043
2	7.326638	-0.846723	-6.698854	-0.896532	0.154282
3	-4.417806	-0.035612	-5.089965	-1.116532	0.065393
4	1.504416	-0.967834	11.737813	2.389023	-0.201274

Relative Difference

Features / Clusters	SHOT'CLOCK	TOUCH'TIME	SHOT'DIST	CLOSE'DEF'DIST	SHOT'RESULT
0	-10.371720	276.586252	-51.338818	-35.536361	-9.379509
1	-46.103604	-12.012298	90.743251	34.143140	-32.905983
2	54.640045	-33.629030	-67.400666	-23.545593	38.447972
3	-32.946777	-1.414397	-51.212794	-29.323445	16.296296
4	11.219520	-38.439160	118.100273	62.742835	-50.158730

Categorical Interpretation

Features / Clusters	SHOT'CLOCK	TOUCH'TIME	SHOT'DIST	CLOSE'DEF'DIST	SHOT'RESULT
0	None	much bigger	much smaller	smaller	None
1	smaller	None	much bigger	None	None
2	much bigger	None	much smaller	None	bigger
3	None	None	much smaller	None	None
4	None	smaller	much bigger	much bigger	much smaller

- 0-th cluster consists of shots made from under the basket with a lot of dribbling and resistance from the opposite team;
- the first cluster consists of shots made right before a shot clock buzzer afar from baskets;
- 2-d cluster represents shots made from under the basket right after a ball changed sides(restart of the shot clock);
- 3-d cluster just shots near the basket;
- 4-th cluster represents mostly unsuccessful shots made from afar right after obtaining the ball.

9-cluster partition

```
In [12]: means, a_diff, r_diff, mask = describe_cluster(int_data,km9.labels_)
         print(means,'\n')
         print(a_diff,'\n')
         print(r_diff,'\n')
         print(mask)
```

gives us:

Cluster Mean

Clusters \ Features	SHOT˙CLOCK	TOUCH˙TIME	SHOT˙DIST	CLOSE˙DEF˙DIST	SHOT˙RESULT
0	8.780000	2.597143	4.148571	2.308571	0.428571
1	16.386667	1.206667	23.933333	5.693333	0.266667
2	23.338095	0.961905	2.142857	2.114286	0.523810
3	7.800000	1.341667	23.683333	5.000000	0.250000
4	5.720000	2.960000	13.120000	4.400000	0.400000
5	12.800000	2.443750	17.500000	4.981250	0.187500
6	11.075000	1.350000	20.550000	12.450000	0.000000
7	17.657692	1.753846	3.800000	3.746154	0.615385
8	13.415385	8.884615	5.323077	2.638462	0.384615
grand	13.408917	2.517834	9.938854	3.807643	0.401274

Absolute Difference

Clusters \ Features	SHOT_CLOCK	TOUCH_TIME	SHOT_DIST	CLOSE_DEF_DIST	SHOT_RESULT
0	-4.628917	0.079308	-5.790282	-1.499072	0.027298
1	2.977749	-1.311168	13.994480	1.885690	-0.134607
2	9.929178	-1.555930	-7.795996	-1.693358	0.122536
3	-5.608917	-1.176168	13.744480	1.192357	-0.151274
4	-7.688917	0.442166	3.181146	0.592357	-0.001274
5	-0.608917	-0.074084	7.561146	1.173607	-0.213774
6	-2.333917	-1.167834	10.611146	8.642357	-0.401274
7	4.248775	-0.763988	-6.138854	-0.061489	0.214111
8	0.006467	6.366781	-4.615777	-1.169182	-0.016659

Relative Difference

Clusters \ Features	SHOT_CLOCK	TOUCH_TIME	SHOT_DIST	CLOSE_DEF_DIST	SHOT_RESULT
0	-34.521186	3.149868	-58.259054	-39.370071	6.802721
1	22.207233	-52.075217	140.805776	49.523810	-33.544974
2	74.049067	-61.796345	-78.439594	-44.472590	30.536659
3	-41.829755	-46.713467	138.290396	31.314821	-37.698413
4	-57.341820	17.561346	32.007178	15.557042	-0.317460
5	-4.541136	-2.942386	76.076647	30.822390	-53.273810
6	-17.405710	-46.382494	106.764291	226.973904	-100.000000
7	31.686191	-30.343070	-61.766214	-1.614896	53.357753
8	0.048232	252.867345	-46.441741	-30.706179	-4.151404

Categorical Interpretation

Clusters \ Features	SHOT_CLOCK	TOUCH_TIME	SHOT_DIST	CLOSE_DEF_DIST	SHOT_RESULT
0	None	None	much smaller	smaller	None
1	None	much smaller	much bigger	bigger	None
2	much bigger	much smaller	much smaller	smaller	None
3	smaller	smaller	much bigger	None	smaller
4	much smaller	None	None	None	None
5	None	None	much bigger	None	much smaller
6	None	smaller	much bigger	much bigger	much smaller
7	None	None	much smaller	None	much bigger
8	None	much bigger	smaller	None	None

Here we do not observe many interesting clusters. For example, 5-th is more average version of 6-th. and 6-th cluster is similar to 4-th cluster from 5-cluster partition Here mostly successful shot got in 7-th cluster with short distance to basket. 2-th cluster may describe fast plays right after a tip off when a shooting player dashes to the basket and shots after receiving a pass.

We don't see a lot of interesting clusters in both partitions but we find 5-cluster partition more helpful because it is obviously smaller and includes interesting cluster from 9-cluster partition as well.

2.4 Bootstrap Mean Estimation

Here we define a set of functions to help us with bootstrapping data and estimation of means. The `bootstrap` function returns `n_samples` bootstrapped means of given `feature` conditional on `cluster` in partition given by `labels`. `pivotal_estimate` and `non_pivotal_estimate` compute mean estimate of given `feature` and its `(1-alpha)`-confidence interval. `compare_cluster` compares `feature` between to clusters given by `clusters` by estimating mean of their difference and quotients. If only one cluster is given function will compare it with whole dataset.

```
In [69]: def bootstrap(data,labels,cluster,feature,n_samples=5000):

             X = data[labels == cluster][feature].sample(frac=n_samples,replace=True)
             X = pd.DataFrame(X)
             X['sample'] = np.repeat(range(n_samples),(labels == cluster).sum())

             means = X.groupby('sample')[feature].mean()
             return means

         def pivotal_estimate(data,labels,cluster,
                              feature,n_samples=5000,alpha=0.05):
             means = bootstrap(data,labels,cluster,feature,n_samples)
             mean = means.mean()
             std = means.std()

             # theoretical quantile of standard normal distribution
             quantile  = sc.stats.norm.ppf(1-alpha/2)
             lb = mean - quantile*std
             rb = mean + quantile*std
             return({'mean':mean,'boundaries':(lb,rb)})

         def non_pivotal_estimate(data,labels,cluster,
                                  feature,n_samples=5000,alpha=0.05):
             means = bootstrap(data,labels,cluster,feature,n_samples)
             mean = means.mean()
             # empirical quantiles of set of means:
             lb = mquantiles(means,alpha/2)[0]
             rb = mquantiles(means,1-alpha/2)[0]

             return({'mean':mean,'boundaries':(lb,rb)})

         def compare_clusters(data, labels, clusters,
                              feature, n_samples=5000,alpha=0.05):
```

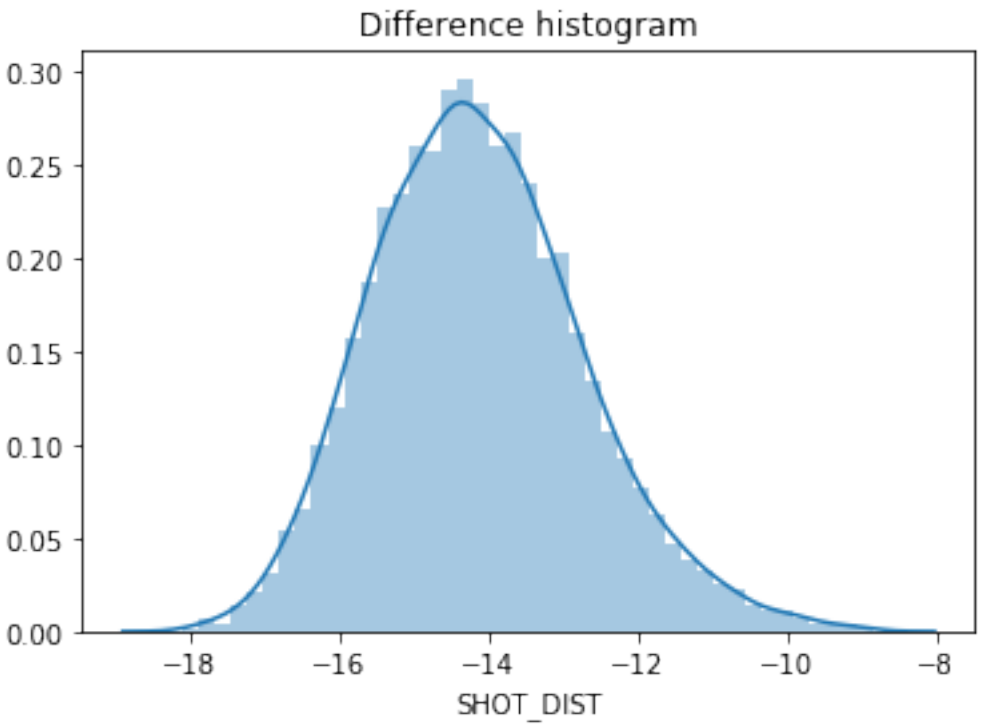
```
    m1 = bootstrap(data,labels,clusters[0],feature,n_samples)
    if len(clusters) == 1:
        m2 = bootstrap(data,np.ones(data.shape[0]),1,feature,n_samples)
    else:
        m2 = bootstrap(data,labels,clusters[1],feature,n_samples)
    d = m1-m2
    q = m1/m2
    sns.distplot(m1-m2).set_title(f'Difference histogram')
    d_mean = d.mean()
    d_std = d.std()
    q_mean = q.mean()
    q_std = q.std()
    quantile  = sc.stats.norm.ppf(1-alpha/2)
    d_pivotal = d_mean - quantile*d_std, d_mean + quantile*d_std
    q_pivotal = q_mean - quantile*q_std, q_mean + quantile*q_std
    d_non_pivotal =  mquantiles(d,alpha/2)[0], mquantiles(d,1-alpha/2)[0]
    q_non_pivotal =  mquantiles(q,alpha/2)[0], mquantiles(q,1-alpha/2)[0]
    return {'Difference':
            {'mean':d_mean,
             'pivotal boundaries':d_pivotal,
             'non-pivotal boundaries':d_non_pivotal
            },
            'Quotient':
            {'mean':q_mean,
             'pivotal boundaries':q_pivotal,
             'non-pivotal boundaries': q_non_pivotal
            }
           }
```

Comparison of two clusters

Here we use `compare_clusters` function to compare `SHOT_DIST` feature between 0-th and 1-th cluster in 5-cluster partition

```
In [70]: compare_clusters(match_data,km5.labels_,(0,1),'SHOT_DIST')
```

```
Out[70]: {'Difference': {'mean': -14.12831734265734,
           'pivotal boundaries': (-16.896649409752023, -11.359985275562657),
           'non-pivotal boundaries': (-16.64776048951049, -11.056660839160838)},
          'Quotient': {'mean': 0.25511941543787153,
           'pivotal boundaries': (0.13370733116164135, 0.3765314997141017),
           'non-pivotal boundaries': (0.16057219155075486, 0.39527017397100817)}}
```



We see that difference of SHOT_DIST between 0-th and 1-st clusters a greatly less than zero, and the corresponding quotient is less than one. So we can conclude that SHOT_DISTin 0-th cluster are less than in first.

Grand mean estimation

Here we obtain pivotal and non-pivotal confidence intervals for SHOT_DIST grand mean

```
In [73]: print('Pivotal grand mean:')
         pivotal_estimate(match_data,np.ones(match_data.shape[0]),1,'SHOT_DIST')
```

```
Pivotal grand mean:
```

```
Out[73]: {'mean': 9.928289936305735,
          'boundaries': (8.607661422858309, 11.24891844975316)}
```

```
In [74]: print('Non-pivotal grand mean:')
         non_pivotal_estimate(match_data,np.ones(match_data.shape[0]),1,'SHOT_DIST')
```

```
Non-pivotal grand mean:
```

```
Out[74]: {'mean': 9.938222675159238,
          'boundaries': (8.604337579617837, 11.317439490445869)}
```

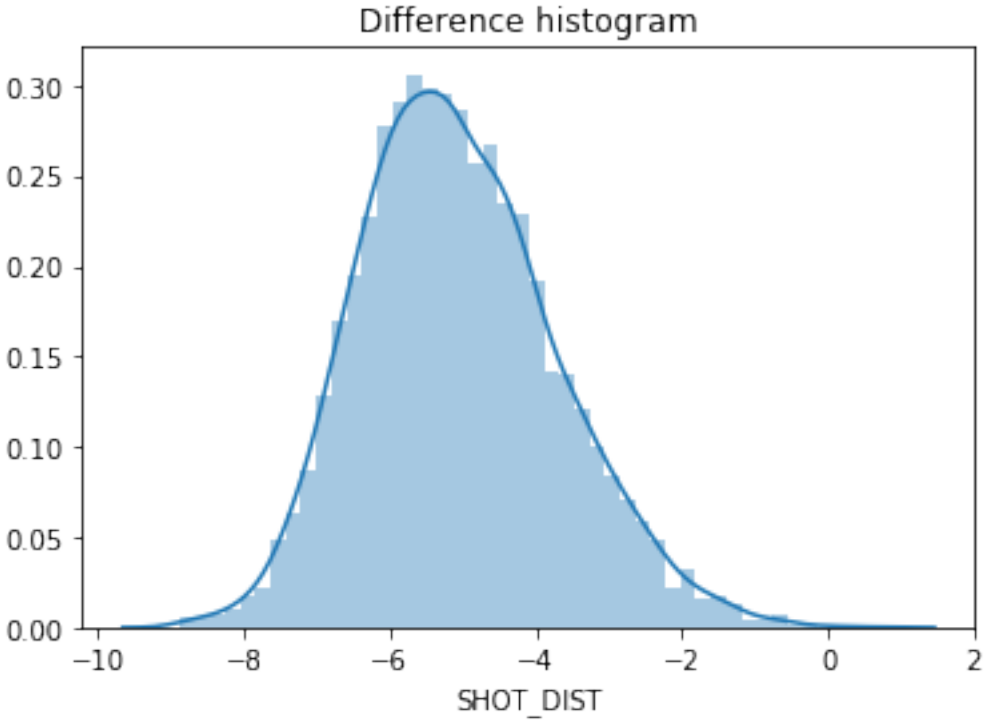
We see that obtained intervals are somewhat the same.

Comparison of cluster mean with grand mean

To compare SHOT_DIST's mean in zeroth cluster with corresponding grand mean we again apply comapre_clusters function.

```
In [72]: compare_clusters(match_data,km5.labels_,[0],'SHOT_DIST')
```

```
Out[72]: {'Difference': {'mean': -5.060902339316736,
   'pivotal boundaries': (-7.693323718099329, -2.4284809605341433),
   'non-pivotal boundaries': (-7.3903045744064855, -2.214817313259994)},
  'Quotient': {'mean': 0.49256161269359783,
   'pivotal boundaries': (0.2528264432220447, 0.7322967821651509),
   'non-pivotal boundaries': (0.3018234660453669, 0.7724986044645963)}}
```



Again both confidence intervals for difference lay beneath zero. And quotient is less than one. So we conclude that grand mean of SHOT_DIST is greater than its mean in zeroth cluster.

3 Contingency Table

3.1 Feature Selection

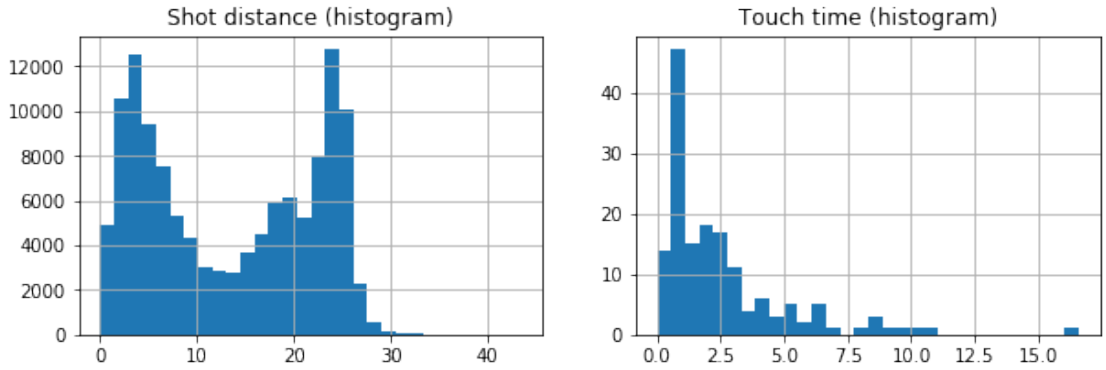
We will consider three features:

1. **SHOT_RESULT** - whether the shot was successful.
2. **SHOT_DIST** - the distance between the shooting player and the basket before making the shot.
3. **TOUCH_TIME** - how long the shooting player has possession prior to taking the shot.

The first feature is binary, so it can be already considered as nominal. This feature is particularly interesting in the context of basketball statistics, because knowledge of how the shot result is related to other conditions may give a clue about how and when to make better shots.

The second and third features are quantitative. We will transform them into nominal by splitting the ranges of their values into several chunks. The local minima in the corresponding histograms define the border between some of the chunks:

```
In [3]: plt.figure(figsize=(10, 3))
        plt.subplot(121)
        X['SHOT_DIST'].hist(bins=30)
        plt.title('Shot distance (histogram)')
        plt.subplot(122)
        X['TOUCH_TIME'].hist(bins=30)
        plt.title('Touch time (histogram)');
```




```
In [4]: def choose_bin(bins, value):
            for i in range(1, len(bins)):
                if bins[i - 1] <= value < bins[i]:
                    return '{}..{}'.format(bins[i - 1], bins[i])
            raise ValueError('No appropriate bin')

        X['SHOT_DIST_CAT'] = X['SHOT_DIST'].apply(
            lambda x: choose_bin([0, 12, 22, 40], x))
        X['TOUCH_TIME_CAT'] = X['TOUCH_TIME'].apply(
            lambda x: choose_bin([0, 3, 7, 20], x))
```

The range of **SHOT_DIST** was split into 3 parts: $[0, 12), [12, 22), [22, 40)$.
The range of **TOUCH_TIME** was also split into 3 parts: $[0, 3), [3, 7), [7, 20)$.

3.2 Conditional Frequency and Quetelet Relative Index Tables

We present the conditional frequency table and the Quetelet relative index table for the data:

```
In [5]: display(pd.crosstab(X['SHOT_RESULT'], X['SHOT_DIST_CAT'],
                            margins=True, margins_name='Total'))
        display(pd.crosstab(X['SHOT_RESULT'], X['TOUCH_TIME_CAT'],
                            margins=True, margins_name='Total'))
```

Conditional Frequencies

Result \ Distance	0..12	12..22	22..40	Total
0 (miss)	51	27	16	94
1 (success)	49	8	6	63
Total	100	35	22	157

Result \ Touch time	0..3	3..7	7..20	Total
0 (miss)	67	22	5	94
1 (success)	49	10	4	63
Total	116	32	9	157

```
In [6]: def quetelet_table(dataframe, ver_feature, hor_feature):
            ver_column = dataframe[ver_feature]
            hor_column = dataframe[hor_feature]
            table = defaultdict(dict)
            for ver_value in sorted(ver_column.unique()):
                for hor_value in sorted(hor_column.unique()):
                    p_vk = ((ver_column == ver_value) &
                            (hor_column == hor_value)).mean()
                    p_k = (ver_column == ver_value).mean()
                    p_v = (hor_column == hor_value).mean()

                    quetelet_index = p_vk / (p_k * p_v) - 1
                    table[hor_value][ver_value] = quetelet_index

            df = pd.DataFrame(table,
                              index=sorted(ver_column.unique()),
                              columns=sorted(hor_column.unique()))
            df = df.rename_axis(ver_feature, axis=0)
            df = df.rename_axis(hor_feature, axis=1)
            return df
```

```
display(quetelet_table(X, 'SHOT_RESULT', 'SHOT_DIST_CAT'))
display(quetelet_table(X, 'SHOT_RESULT', 'TOUCH_TIME_CAT'))
```

Quetelet Relative Indices

Result \ Distance	0..12	12..22	22..40
0 (miss)	-0.148191	0.288450	0.214700
1 (success)	0.221111	-0.430385	-0.320346

Result \ Touch time	0..3	3..7	7..20
0 (miss)	-0.035308	0.148271	-0.072104
1 (success)	0.052682	-0.221230	0.107584

From the conditional frequency table, we can see that most shots are not successful, most shots are made from the small distances and with low touch times.

The Quetelet relative index table provides insight into how the values may depend on each other. In the first table, we can see that a cell corresponding to successful shots from small distances contains a positive value, while cells corresponding to larger distances contains negative values. This suggests that shots made from the small distances are more likely to be successful (which coincides with our intuition about basketball).

Compared with the first table, the second one contains smaller values. This suggests that the shot result depends much less on the touch time. The signs of values suggest that the shots with very small and very large touch time are more likely to be successful:

- If the touch time is small, enemy's defenders don't have time to react to a quick pass of the ball.
- If the touch time is large, this may mean that there were no defenders around, so the player had time to find the right position and make a shot.

Though we provided a possible explanation for the results, there is no strong evidence that this dependence indeed exists (as we will see later from X^2 value).

3.3 Chi-square / Summary Quetelet Index

Let's calculate the value of X^2 (or summary Quetelet index Q) over both tables:

```
In [7]: def chi_square(dataframe, ver_feature, hor_feature):
            ver_column = dataframe[ver_feature]
            hor_column = dataframe[hor_feature]
            result = 0
            for ver_value in sorted(ver_column.unique()):
```

```
                for hor_value in sorted(hor_column.unique()):
                    p_vk = ((ver_column == ver_value) &
                            (hor_column == hor_value)).mean()
                    p_k = (ver_column == ver_value).mean()
                    p_v = (hor_column == hor_value).mean()

                    result += (p_vk - p_k * p_v)**2 / (p_k * p_v)

            max_result = min(len(ver_column.unique()), len(hor_column.unique())) - 1

            return result, max_result

        print('X^2 for SHOT_RESULT and SHOT_DIST_CAT: {:.4f} (max {:.1f})'.format(
            *chi_square(X, 'SHOT_RESULT', 'SHOT_DIST_CAT')))
        print('X^2 for SHOT_RESULT and TOUCH_TIME_CAT: {:.4f} (max {:.1f})'.format(
            *chi_square(X, 'SHOT_RESULT', 'TOUCH_TIME_CAT')))
```

```
X^2 for SHOT_RESULT and SHOT_DIST_CAT: 0.0582 (max 1.0)
X^2 for SHOT_RESULT and TOUCH_TIME_CAT: 0.0085 (max 1.0)
```

We can see that the possible dependence between the values is small. There is much more evidence that the dependence exists in the first case.

3.4 Desired Number of Observations

Let's calculate the number of observations that would suffice to see the features as associated. Remember that if the hypothesis of independence is true, then:

$NX^2 \sim \chi^2((K-1)(L-1))$, where K, L are the numbers of possible different values for the features

In this case, we have 2 degrees of freedom:

```
In [8]: print(chi2(df=2).ppf(0.95))
        print(chi2(df=2).ppf(0.99))
```

```
5.99146454710798
9.21034037197618
```

If the features are independent, there is only 5% chance that NX^2 will be greater than 5.99, and 1% chance that NX^2 will be greater than 9.21.

We want to find such N that NX^2 will exceed specified values, so we will calculate $\frac{5.99}{X^2}$ and $\frac{9.21}{X^2}$:

```
In [9]: feature1 = 'SHOT_RESULT'
        for feature2 in ['SHOT_DIST_CAT', 'TOUCH_TIME_CAT']:
            for confidence in [0.95, 0.99]:
                min_N = (chi2(df=2).ppf(confidence) /
                         chi_square(X, feature1, feature2)[0])
                print('{} and {} are associated with confidence {:.2f}\n'
                      '\twhen N >= {:.1f}'.format(
                    feature1, feature2, confidence, min_N))

SHOT_RESULT and SHOT_DIST_CAT are associated with confidence 0.95
        when N >= 103.0
SHOT_RESULT and SHOT_DIST_CAT are associated with confidence 0.99
        when N >= 158.3
SHOT_RESULT and TOUCH_TIME_CAT are associated with confidence 0.95
        when N >= 704.5
SHOT_RESULT and TOUCH_TIME_CAT are associated with confidence 0.99
        when N >= 1083.0
```

In our case, $N = 157$. We can conclude that we have enough data to say that the shot result depends on the shot distance with 95% confidence (and almost enough data for 99% confidence).

Still, there is not enough data to say that the shot result depends on the touch time.

4 Principal Component Analysis

4.1 Feature Selection

We will consider the following features:

1. **SHOT_CLOCK** - time until the shot clock violation
2. **TOUCH_TIME** - how long the shooting player has been in possession of the ball before the shot
3. **SHOT_DIST** - the distance between the shooting player and the basket before making the shot.

These features intuitively should have the most influence on the result of the shot compared to the other per-shot numeric features. For example, as demonstrated before, there is a simple dependency between **DRIBBLES** and **TOUCH_TIME**, and **GAME_CLOCK** is unlikely to have an influence large enough to be detected and interpreted using PCA.

Also we will color the points in the visualizations according to **SHOT_RESULT**, i.e. whether the shot was successful or not. This is a natural grouping of the points, as making successful shots is the main point of the game.

4.2 Performing the PCA

Let us standardize the dataset and compute its data scatter:

```
In [214]: pca_data = match_data[['SHOT_CLOCK', 'TOUCH_TIME', 'SHOT_DIST']]
          pca_std = (pca_data - pca_data.mean()) / pca_data.std()

          print('Data scatter:', (pca_data ** 2).sum().sum())
          print('Data scatter after centering:',
                ((pca_data - pca_data.mean()) ** 2).sum().sum())
          print('Data scatter after standardization:',
                (pca_std ** 2).sum().sum())
```

```
Data scatter: 62902.23000000001
Data scatter after centering: 18169.89057324842
Data scatter after standardization: 467.99999999999943
```

Let us now perform the PCA using the PCA class from scikit-learn:

```
In [241]: pca = PCA()
          transformed_array = pca.fit_transform(pca_std)
          transformed = pd.DataFrame(transformed_array, index=pca_data.index,
                                     columns=('PC1', 'PC2', 'PC3'))

          print('Transformed:')
          print(transformed.head())
          print('\t...')

          print('\nComponents:')
          print(pca.components_)
          print()

          scatter_after_std = (pca_std ** 2).sum().sum()
          for col_name in transformed:
              scatter = (transformed[col_name] ** 2).sum()
              scatter_percent = scatter / scatter_after_std * 100
              print(f'{col_name} contributes {scatter:.3f}, or'
                    f' {scatter_percent:.2f}%, to the data scatter')
```

```
Transformed:
             PC1       PC2       PC3
27511 -0.349038 -0.053573  1.003101
27512 -0.509086  0.753252  0.571980
```

```
27513 -0.583496  0.072316  1.214208
27514 -1.683608 -0.338379  0.016881
27515 -0.995870 -0.089864  0.605426
        ...

Components:
[[-0.77237014  0.28342248  0.56843299]
 [-0.14022404  0.7967612  -0.58779997]
 [-0.61950108 -0.53370711 -0.57565192]]

PC1 contributes 197.840, or 42.27%, to the data scatter
PC2 contributes 180.450, or 38.56%, to the data scatter
PC3 contributes 89.710, or 19.17%, to the data scatter
```

The variances explained by the first and second components are not stellar, but PCA still managed to perform at least a somewhat meaningful transformation.

4.3 Visualization

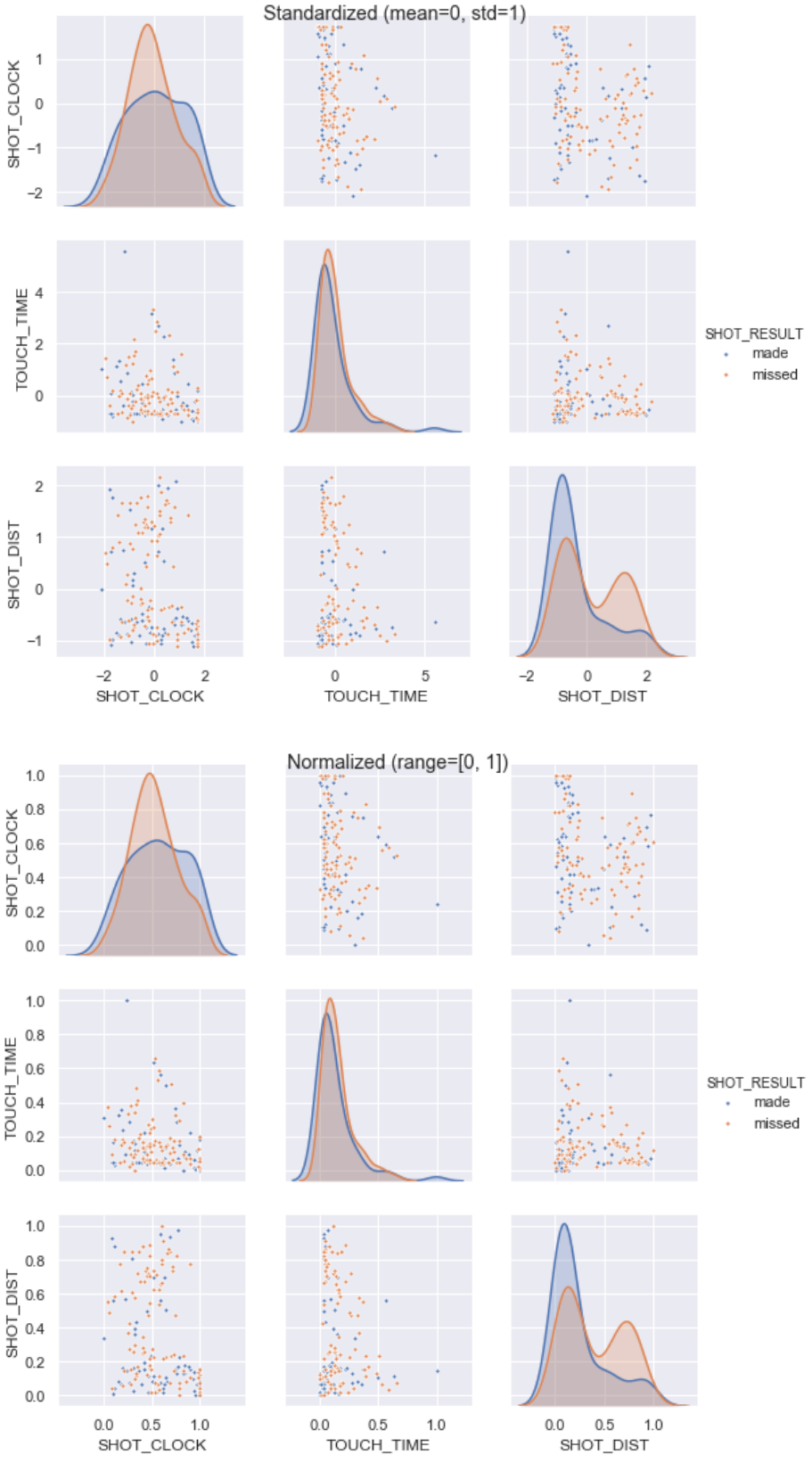
Let us now visualize the data before and after PCA in the form of pairwise scatter plots and per-feature line plots. To assess the helpfulness of the PCA we will append the **SHOT_RESULT** column to the data and color the dots and the lines according to whether the shot was successful. In addition, let us normalize the features into the $[0, 1]$ range and plot the result as well:

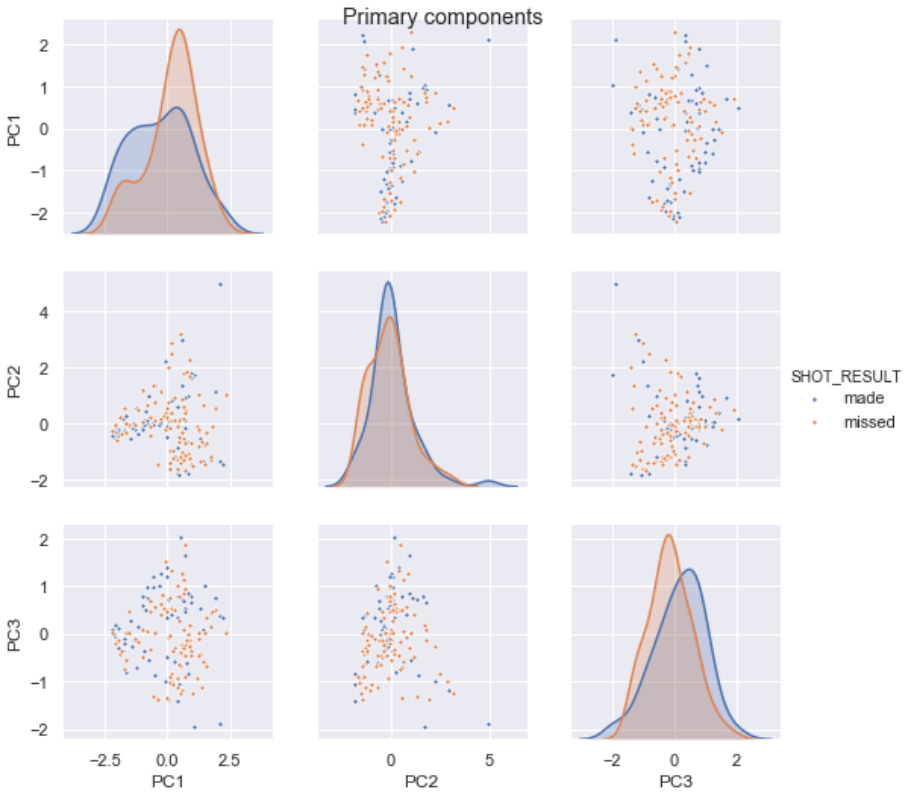
```
In [242]: pca_01 = (pca_data - pca_data.min()) \
                  / (pca_data.max() - pca_data.min())

          for df in (pca_std, pca_01, transformed):
              df['SHOT_RESULT'] = match_data['SHOT_RESULT']

          def pairplot(df, title):
              plot = sns.pairplot(df, hue='SHOT_RESULT', plot_kws=dict(s=10))
              plot.fig.suptitle(title)

          pairplot(pca_std, 'Standardized (mean=0, std=1)')
          pairplot(pca_01, 'Normalized (range=[0, 1])')
          pairplot(transformed, 'Primary components')
```


Standardized (mean=0, std=1)
SHOT_CLOCK
TOUCH_TIME
SHOT_DIST
SHOT_RESULT
made
missed
Normalized (range=[0, 1])
SHOT_CLOCK
TOUCH_TIME
SHOT_DIST
SHOT_RESULT
made
missed



Obviously, since we only use line plots and pairwise scatter plots, the difference between standardization and normalization amounts to relabeling of the axes (which might be helpful by itself because it makes interpreting the coordinates in the graph conceptually easier). However, if we were, for example, using a 3D scatter plot with fixed scales of the axes, normalization into the $[0, 1]$ range would help a lot as it guarantees that the features have the same scale (if there are no outliers).

PCA doesn't seem to help a lot with distinguishing the points with different shot results: both before and after PCA features aren't nearly linearly separable. This is unsurprising considering the low values of the explained variance. However, there does seem to be a slight improvement in that if we consider only the values of the first primary component, successful shots are on average to the left on the graph.

4.4 Hidden factor extraction

To obtain a hidden factor expressed in the 0-100 rank scale, we first rescale all the features to this range and then decompose the result using SVD and output the first component. Intuition should tell us that **SHOT_DIST** correlates negatively with the shot successfulness, while **SHOT_CLOCK** and **TOUCH_TIME** correlate with it positively (players shoot better from short range, with more time left on the shot clock and after holding the ball

longer), so multiplying **SHOT_DIST** by -1 should ensure that the feature loadings are positive. It is the case empirically as well:

```
In [321]: def rescale(df, low=0, high=1):
              return (df - df.min()) / (df.max() - df.min()) * (high - low) + low

          rescaled_pca_data = pca_data.copy()
          rescaled_pca_data.SHOT_DIST *= -1
          rescaled_pca_data = rescale(rescaled_pca_data, 0, 100)

          U, s, V = sla.svd(rescaled_pca_data)
          U *= -1
          V *= -1
          contribution = 100 * s[0] ** 2 / (rescaled_pca_data ** 2).sum().sum()
          print('First component:', V[0])
          print(f'Its contribution to the data scatter: {contribution:.3f}%')
```

```
First component: [0.62131743 0.15860363 0.76733926]
Its contribution to the data scatter: 91.167%
```

That the contribution of the first component is much higher than in the previous case should not be surprising: the data is not centered, so its mean is the source of most of the data scatter.

The value of the first component z is $0.6213 \cdot$ **SHOT_CLOCK** $+ 0.1586 \cdot$ **TOUCH_TIME** $+ 0.7673 \cdot$ **SHOT_DIST**. To ensure that the hidden factor is inside $[0, 100]$, let us set α such that $\alpha \cdot (0.6213 \cdot 100 + 0.1586 \cdot 100 + 0.7673 \cdot 100) = 100$. Then $\alpha = (0.6213 + 0.1586 + 0.7673)^{-1} = 0.6463$ and the corresponding score vector is computed as $\alpha z = 0.4016 \cdot$ **SHOT_CLOCK** $+ 0.1025 \cdot$ **TOUCH_TIME** $+ 0.4959 \cdot$ **SHOT_DIST**. (Keep in mind that the **SHOT_DIST** scale is inverted here.)

This hidden factor expression seems to imply that most of the variation in the players' shots comes from their distance to the ring and then from how much they hurry because of the shot clock. How much the ball was in possession before the shot seems to factor in much less. Thus, it can be argued that the first hidden factor we found is the combination of the technical difficulty of the shot due to distance and the tactical difficulty due to the lack of time.

5 Linear Regression

5.1 Feature Selection

We will consider two features:

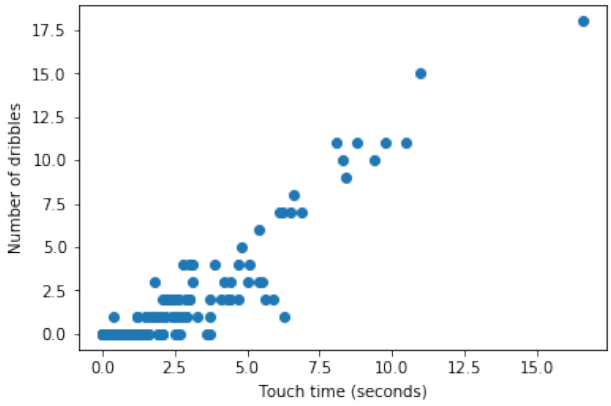
1. **TOUCH_TIME** - how long the shooting player has possession prior to taking the shot.

2. **DRIBBLES** - a number of dribbles made by the shooting player.

Dribbling is the only way of how a player may possess the ball in basketball, so it is natural to expect that the number of dribbles will be roughly proportional to the possession time. We can see this at the scatter plot:

```
In [10]: reg_x = X['TOUCH_TIME'].values
         reg_y = X['DRIBBLES'].values

         plt.scatter(reg_x, reg_y)
         plt.xlabel('Touch time (seconds)')
         plt.ylabel('Number of dribbles');
```



5.2 Regression Model

Let's build a linear regression to model how the number of dribbles depends on the touch time.

The slope corresponds to the average "speed of dribbling" (the number of dribbles a player makes per second).

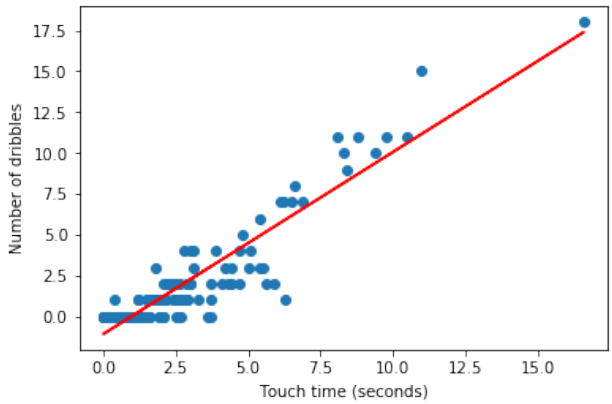
```
In [11]: rho = np.corrcoef(reg_x, reg_y)[0, 1]
         a = rho * reg_y.std() / reg_x.std()
         b = reg_y.mean() - a * reg_x.mean()

         reg_pred = a * reg_x + b

         plt.scatter(reg_x, reg_y)
         plt.plot(reg_x, reg_pred, c='red')
         plt.xlabel('Touch time (seconds)')
         plt.ylabel('Number of dribbles')
         plt.show()
```

```
det_coeff = rho**2

print('Slope: {:.3f}'.format(a))
print('Intercept: {:.3f}'.format(b))
print('Correlation: {:.3f}'.format(rho))
print('Determinancy coeff.: {:.3f}'.format(det_coeff))
```




```
Slope: 1.113
Intercept: -1.084
Correlation: 0.928
Determinancy coeff.: 0.860
```

We can see that a player makes ≈ 1.113 dribbles per second on average.

The correlation between the features is high (0.928). Determinacy coefficient is 0.860, which means that the linear regression explains 86% of the variance in data.

5.3 Example of Predictions

We can use the linear regression model to predict the number of dribbles using the touch time:

```
In [12]: test_x = np.array([10, 15, 20])
         test_pred = a * test_x + b
         print(test_pred)
```

```
[10.0510251  15.61845057 21.18587604]
```

If a player possesses the ball for 10 seconds, he is likely to make ≈ 10.05 dribbles. For 15 seconds, it's ≈ 15.62 dribbles. For 20 seconds, it's ≈ 21.2 dribbles.

5.4 Mean Relative Absolute Error and Determinacy Coefficient

To calculate the mean relative absolute error, we need to remove all samples where the target value (the number of dribbles) is zero.

While the determinacy coefficient is high (86% of the variance explained), it turns out that the mean relative absolute error of the predictions is close to 51%:

```
In [13]: pred = reg_pred[reg_y != 0]
         y = reg_y[reg_y != 0]
         rel_mae = np.abs((pred - y) / y).mean()
         print('Mean relative absolute error: {:.3f}'.format(rel_mae))

         mae = np.abs(pred - y).mean()
         print('Mean absolute error: {:.3f}'.format(mae))

Mean relative absolute error: 0.509
Mean absolute error: 1.044
```

We got this result because the target variable contains a lot of small values (1-2 dribbles). While the mean absolute error of the regression is only ≈ 1 dribble, the relative error is quite high on these examples.

We can conclude that the high determinacy coefficient does not always mean that the relative error on data will be low. The obtained regression model is quite accurate in case of large touch times but gives large relative errors on low touch times.