Assignment 1

March 5, 2018

```
In [1]: from IPython.display import HTML
        HTML('''<style>html, body{overflow-y: visible !important} .CodeMirror{min-width:105% !im

Out[1]: <IPython.core.display.HTML object>
```

1 Foundations of Data Mining: Assignment 1

Please complete all assignments in this notebook. You should submit this notebook, as well as a PDF version (See File > Download as).

```
In [2]: # Please fill in your names here
        NAME_STUDENT_1 = "Jan-Niklas Schneider 1260421"
        NAME_STUDENT_2 = "Georgiana Manolache 0876359"
```

```
In [3]: %matplotlib inline
        from preamble import *
        plt.rcParams['savefig.dpi'] = 100 # This controls the size of your figures
        # Comment out and restart notebook if you only want the last output of each cell.
        InteractiveShell.ast_node_interactivity = "all"
```

1.1 MoneyBall (5 points, 1+2+1+1)

In the early 2000s, 2 baseball scouts completely changed the game of baseball by analysing the available data about baseball players and hiring the best ones. The MoneyBall dataset contains this data (click the link for more details). The goal is to accurately predict the number of 'runs' each player can score.

```
In [4]: moneyball = oml.datasets.get_dataset(41021) # Download MoneyBall data
        # Get the predictors X and the target y
        X, y, attribute_names = moneyball.get_data(target=moneyball.default_target_attribute, re
        # Describe the data with pandas, just to get an overview
        ballframe = pd.DataFrame(X, columns=attribute_names)
        ballframe.describe()

Out[4]:           Team  League     Year       RA   ...    RankPlayoffs        G  \
        count  1232.00  1232.0  1232.00  1232.00   ...          244.00  1232.00
        mean     15.67     0.5  1988.96   715.08   ...            1.72     3.92
```

```
std         9.72     0.5    14.82    93.08   ...              1.10     0.62
min         0.00     0.0  1962.00   472.00   ...              0.00     0.00
25%         7.00     0.0  1976.75   649.75   ...              1.00     4.00
50%        16.00     0.5  1989.00   709.00   ...              2.00     4.00
75%        23.00     1.0  2002.00   774.25   ...              3.00     4.00
max        38.00     1.0  2012.00  1103.00   ...              4.00     7.00

         OOBP    OSLG
count  420.00  420.00
mean     0.33    0.42
std      0.02    0.03
min      0.29    0.35
25%      0.32    0.40
50%      0.33    0.42
75%      0.34    0.44
max      0.38    0.50

[8 rows x 14 columns]
```

1 . Visually explore the data. Plot the distribution of each feature (e.g. histograms), as well as the target. Visualize the dependency of the target on each feature (use a 2d scatter plot). Is there anything that stands out? Is there something that you think might require special treatment? - Feel free to create additional plots that help you understand the data - Only visualize the data, you don't need to change it (yet)

Lets just have a look at some details of the data frame before actually visualize sutff

```
In [5]: print(attribute_names)
```

```
['Team', 'League', 'Year', 'RA', 'W', 'OBP', 'SLG', 'BA', 'Playoffs', 'RankSeason', 'RankPlayoff
```

```
In [6]: print(ballframe.shape)
```

```
(1232, 14)
```

```
In [7]: ballframe.dtypes
```

```
Out[7]: Team             float32
        League           float32
        Year             float32
        RA               float32
        W                float32
        OBP              float32
        SLG              float32
        BA               float32
        Playoffs         float32
        RankSeason       float32
        RankPlayoffs     float32
```

```
G               float32
OOBP            float32
OSLG            float32
dtype: object
```

```
In [8]: with pd.option_context('display.max_rows', None, 'display.max_columns', 15):
            print(ballframe.head(10))
```

```
   Team  League    Year     RA     W   OBP   SLG    BA  Playoffs  RankSeason  \
0   0.0     1.0  2012.0  688.0  81.0  0.33  0.42  0.26       0.0         NaN
1   1.0     1.0  2012.0  600.0  94.0  0.32  0.39  0.25       1.0         3.0
2   2.0     0.0  2012.0  705.0  93.0  0.31  0.42  0.25       1.0         4.0
3   3.0     0.0  2012.0  806.0  69.0  0.31  0.41  0.26       0.0         NaN
4   4.0     1.0  2012.0  759.0  61.0  0.30  0.38  0.24       0.0         NaN
5   5.0     0.0  2012.0  676.0  85.0  0.32  0.42  0.25       0.0         NaN
6   6.0     1.0  2012.0  588.0  97.0  0.31  0.41  0.25       1.0         1.0
7   7.0     0.0  2012.0  845.0  68.0  0.32  0.38  0.25       0.0         NaN
8   8.0     1.0  2012.0  890.0  64.0  0.33  0.44  0.27       0.0         NaN
9   9.0     0.0  2012.0  670.0  88.0  0.34  0.42  0.27       1.0         5.0

   RankPlayoffs    G  OOBP  OSLG
0           NaN  4.0  0.32  0.41
1           4.0  4.0  0.31  0.38
2           3.0  4.0  0.31  0.40
3           NaN  4.0  0.33  0.43
4           NaN  4.0  0.34  0.42
5           NaN  4.0  0.32  0.41
6           3.0  4.0  0.31  0.39
7           NaN  4.0  0.34  0.43
8           NaN  4.0  0.36  0.47
9           1.0  4.0  0.31  0.40
```

Some NaN's are present in RankSeason and RankPlayoffs

```
In [9]: print('targets')
        print(y)
        print("nr of targets {0}".format(len(np.unique(y))))
```

```
targets
[734. 700. 712. ... 878. 774. 599.]
nr of targets 374
```

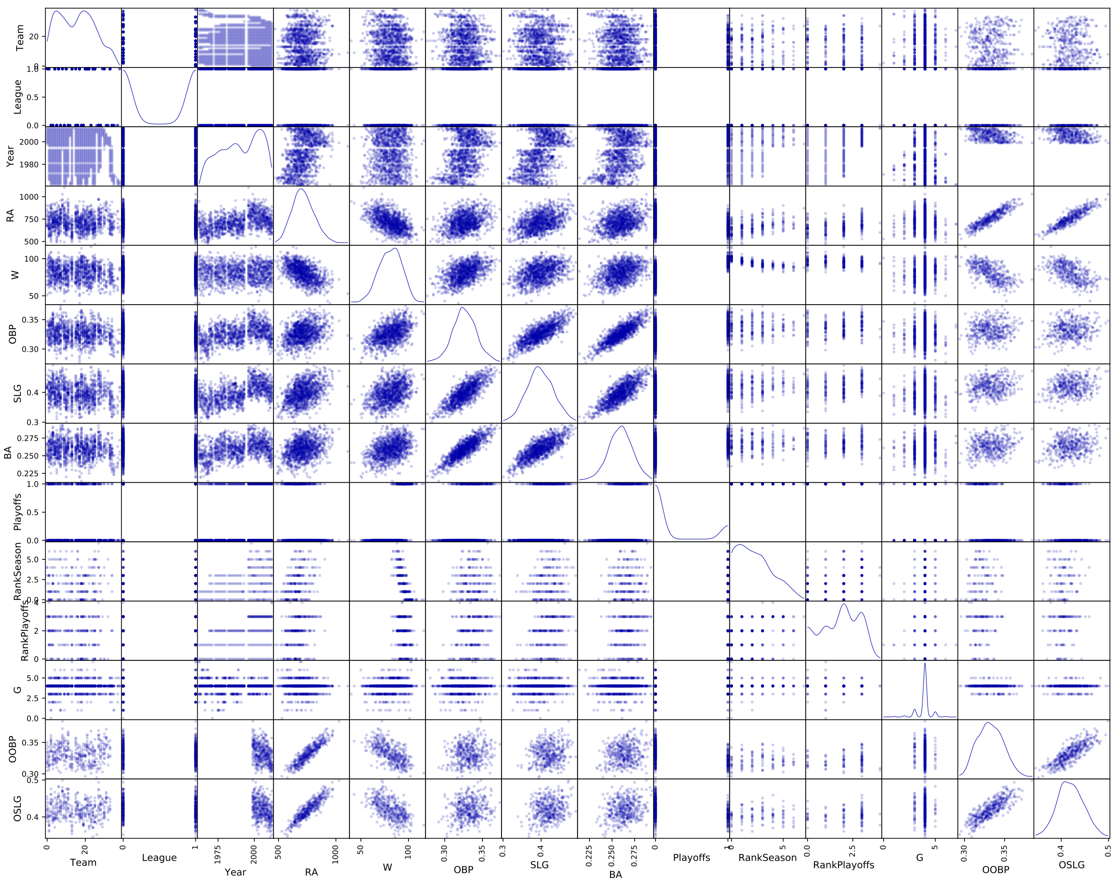
```
In [10]: ballframe['Team'].value_counts()
```

```
Out[10]: 24.0    47
         5.0     47
         7.0     47
```

```
10.0    47
25.0    47
6.0     47
18.0    47
17.0    47
16.0    47
21.0    47
4.0     47
3.0     47
13.0    47
9.0     47
2.0     47
20.0    47
1.0     43
19.0    41
22.0    40
11.0    40
15.0    39
27.0    37
23.0    33
28.0    33
33.0    32
34.0    28
8.0     18
30.0    17
0.0     15
12.0    11
31.0    10
35.0    10
29.0     8
32.0     8
37.0     6
26.0     5
38.0     4
14.0     1
36.0     1
Name: Team, dtype: int64
```

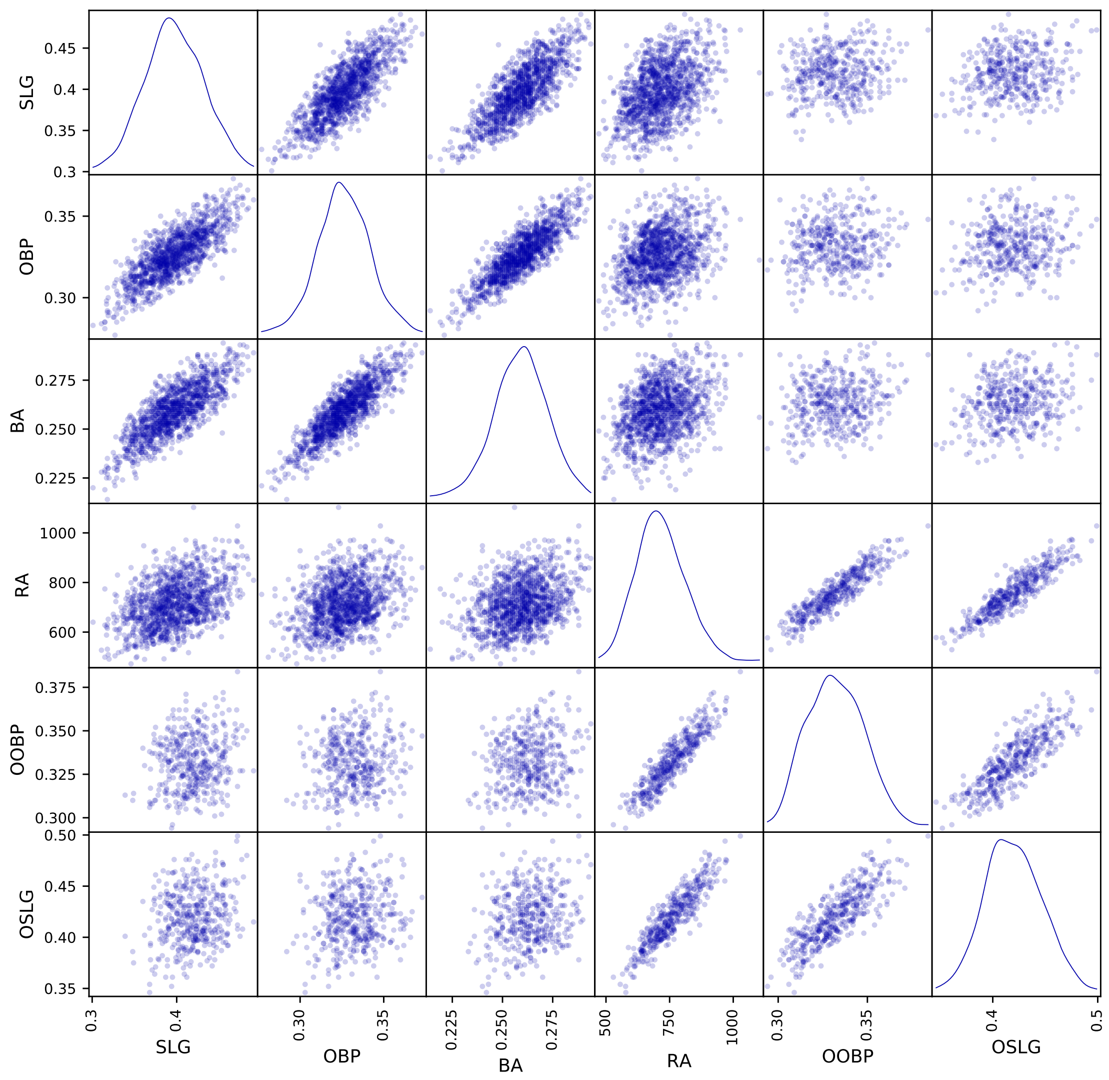
Some teams are more often in data; probably due to having more games in play offs. All data is encoded as floats. What I want to do now is to see whats different between those teams with 47 appearances to the other ones. An Examination of the Moneyball Theory: A Baseball Statistical Analysis this link might be interesting.

```
In [11]: from pandas.tools.plotting import scatter_matrix
         %matplotlib inline
         import matplotlib.pyplot as plt

         p1 = scatter_matrix(ballframe, alpha=0.2, figsize=(20, 16), diagonal='kde')
```



It looks like we have some features that are correlated and seem to draw decent linear function in a 2D environment. These features are SLG - OBP, BA - OBP, RA - OOBP, RA - OSLG, OOBP - OSLG. Lets do some additional plots for those features.

```
In [12]: columns = ['SLG', 'OBP', 'BA', 'RA', 'OOBP', 'OSLG']
         p2 = scatter_matrix(ballframe[columns], alpha=0.2, figsize=(10, 10), diagonal='kde')
```



2 . Compare all linear regression algorithms that we covered in class (Linear Regression, Ridge, Lasso and ElasticNet), as well as kNN. Evaluate using cross-validation and the R^2 score, with the default parameters. Does scaling the data with StandardScaler help? Provide a concise but meaningful interpretation of the results. - Preprocess the data as needed (e.g. are there nominal features that are not ordinal?). If you don't know how to proceed, remove the feature and continue.

```
In [13]: # nan_cols = ['RankSeason', 'RankPlayoffs']
         ballframe = ballframe.fillna(0)
```

Simply filled the missing values for RankSeason and RankPlayoffs by 0 since they seem to be indepent of other variables as observed in the scatter matrices above. Additionally, the runs (target) should not be affected by the season or playoffs.

```
In [14]: from sklearn.model_selection import train_test_split

         X_train, X_test, y_train, y_test = train_test_split(ballframe, y, random_state=0)
```

```
In [15]: print("X_train: {0} ".format(X_train.shape))
         print("X_test: {0} ".format(X_test.shape))
         print("y_train: {0} ".format(y_train.shape))
         print("y_test: {0} ".format(y_test.shape))
         print("For scaled data the shape is the same.")
```

```
X_train: (924, 14)
X_test: (308, 14)
y_train: (924,)
y_test: (308,)
For scaled data the shape is the same.
```

In the assignment it said to also consider also a StandardScaler. I will make use of if for the logistic regression. I am aware that scaling the input will have no noticable effect on the regression. Eventually accuracy will slightly differ because we do a second train/test-split on the day.

```
In [16]: from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
         from sklearn.neighbors import KNeighborsClassifier

         # initialize all regressors and knn
         lr = LinearRegression()
         rid = Ridge()
         lasso = Lasso()
         elastic = ElasticNet()
         knn = KNeighborsClassifier(n_neighbors=3)
```

```
In [17]: from sklearn.preprocessing import StandardScaler

         std_scaler = StandardScaler()
         print(std_scaler)
```

```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

```
In [18]: X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled = train_test_split(Standar
```

```
In [19]: from sklearn.model_selection import cross_val_score
         from sklearn.metrics import r2_score

         # regular data
         lr_scores = cross_val_score(lr, X_train, y_train, cv=5)
         lr.fit(X_train, y_train)
         y_pred_lr = lr.predict(X_test)

         print("Logistic regression test scores:")
         print("cross val scores: {}".format(lr_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_lr)))
         print("Average cv score: {:.2f}".format(lr_scores.mean()))
         print("Variance cv score: {:.4f}".format(np.var(lr_scores)))
```

```
Out[19]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)

Logistic regression test scores:
cross val scores: [0.926 0.936 0.951 0.958 0.95 ]
R^2 score: 0.9513197173125957
Average cv score: 0.94
Variance cv score: 0.0001
```

```
In [20]: # scaled data
         lr_scores = cross_val_score(lr, X_train_scaled, y_train_scaled, cv=5)
         lr.fit(X_train_scaled, y_train_scaled)
         y_pred_lr = lr.predict(X_test_scaled)

         print("Logistic regression test scores with scaled value:")
         print("cross val scores: {}".format(lr_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_lr)))
         print("Average cv score: {:.2f}".format(lr_scores.mean()))
         print("Variance cv score: {:.4f}".format(np.var(lr_scores)))
```

```
Out[20]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)

Logistic regression test scores with scaled value:
cross val scores: [0.926 0.936 0.951 0.958 0.95 ]
R^2 score: 0.9513197510530825
Average cv score: 0.94
Variance cv score: 0.0001
```

```
In [21]: ridge_scores = cross_val_score(rid, X_train, y_train, cv=5)
         rid.fit(X_train, y_train)
         y_pred_ridge = rid.predict(X_test)

         print("Ridge test scores:")
         print("CV scores: {}".format(ridge_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_ridge)))
         print("Average cv score: {:.2f}".format(ridge_scores.mean()))
         print("Variance cv score: {:.4f}".format(np.var(ridge_scores)))
```

```
Out[21]: Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
            normalize=False, random_state=None, solver='auto', tol=0.001)

Ridge test scores:
CV scores: [0.854 0.851 0.897 0.888 0.878]
R^2 score: 0.8980019834547701
Average cv score: 0.87
Variance cv score: 0.0003
```

```
In [22]: lasso_scores = cross_val_score(lasso, X_train, y_train, cv=5)
         lasso.fit(X_train, y_train)
         y_pred_lasso = lasso.predict(X_test)

         print("Lasso test scores:")
         print("CV scores: {}".format(lasso_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_lasso)))
         print("Average cv score: {:.2f}".format(lasso_scores.mean()))
         print("Variance cv score: {:.4f}".format(np.var(lasso_scores)))

Out[22]: Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False)

Lasso test scores:
CV scores: [0.828 0.823 0.877 0.863 0.856]
R^2 score: 0.8736109806742292
Average cv score: 0.85
Variance cv score: 0.0004


In [23]: elastic_scores = cross_val_score(elastic, X_train, y_train, cv=5)
         elastic.fit(X_train, y_train)
         y_pred_elastic = elastic.predict(X_test)

         print("Elastic test scores:")
         print("CV scores: {}".format(elastic_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_elastic)))
         print("Average cv score: {:.2f}".format(elastic_scores.mean()))
         print("Variance cv score: {:.4f}".format(np.var(elastic_scores)))

Out[23]: ElasticNet(alpha=1.0, copy_X=True, fit_intercept=True, l1_ratio=0.5,
               max_iter=1000, normalize=False, positive=False, precompute=False,
               random_state=None, selection='cyclic', tol=0.0001, warm_start=False)

Elastic test scores:
CV scores: [0.828 0.823 0.877 0.863 0.855]
R^2 score: 0.8734690094616266
Average cv score: 0.85
Variance cv score: 0.0004


In [24]: knn_scores = cross_val_score(knn, X_train, y_train, cv=5)
         knn.fit(X_train, y_train)
         y_pred_knn = knn.predict(X_test)

         print("KNN test scores:")
         print("cross val scores: {}".format(knn_scores))
         print("R^2 score: {}".format(r2_score(y_test, y_pred_knn)))
```

```
        print("Average cv score: {:.2f}".format(knn_scores.mean()))
        print("Variance cv score: {:.4f}".format(np.var(knn_scores)))
```

```
C:\Users\Jan\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:605: Warning: The lea
  % (min_groups, self.n_splits)), Warning)
```

```
Out[24]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=3, p=2,
                    weights='uniform')
```

```
KNN test scores:
cross val scores: [0.    0.    0.    0.    0.019]
R^2 score: 0.3666959904517578
Average cv score: 0.00
Variance cv score: 0.0001
```

3 . Do a default, shuffled train-test split and optimize the linear models for the degree of regularization ($alpha$) and choice of penalty (L1/L2). For Ridge and Lasso, plot a curve showing the effect of the training and test set performance (R^2) while increasing the degree of regularization for different penalties. For ElasticNet, plot a heatmap $alpha \times l1\_ratio \rightarrow R^2$ using test set performance. Report the optimal performance. Again, provide a concise but meaningful interpretation. What does the regularization do? Can you get better results? - Think about how you get the L1/L2 loss. This is not a hyperparameter in regression. - We've seen how to generate such heatmaps in Lecture 3.

The first part will look at the optimization of by tuning regularization and penalty by making use of gridsearch and compare to previous default results

```
In [25]: from sklearn.model_selection import GridSearchCV
         from sklearn.pipeline import Pipeline
         from sklearn.model_selection import ShuffleSplit


         lasso_pipe = Pipeline([
             ('estimator', Lasso())
         ])
         params_lasso = [{'estimator__alpha': [0.1,0.25,0.5,0.75,1]}]
         # around 0.01 was doing better so we optimize around this value
         lasso_grid = GridSearchCV(estimator=lasso_pipe, param_grid=params_lasso, cv=ShuffleSpli

In [26]: lasso_grid.fit(X_train, y_train)

Out[26]: GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=None, test_size='default',
                train_size=None),
                error_score='raise',
                estimator=Pipeline(memory=None,
              steps=[('estimator', Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=10
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False))]),
                fit_params=None, iid=True, n_jobs=1,
                param_grid=[{'estimator__alpha': [0.1, 0.25, 0.5, 0.75, 1]}],
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring=None, verbose=0)
```

```
In [27]: print(lasso_grid.best_params_)
         print(lasso_grid.best_score_)
```

```
{'estimator__alpha': 0.1}
0.9240453263934272
```

```
In [28]: lasso_best = lasso_grid.best_estimator_
         y_pred_lasso_best = lasso_best.predict(X_test)

         print("Lasso test scores before optimization:")
         print("R^2 score: {}".format(r2_score(y_test, y_pred_lasso)))

         print("Lasso test scores after optimization:")
         print("R^2 score: {}".format(r2_score(y_test, y_pred_lasso_best)))
```

```
Lasso test scores before optimization:
R^2 score: 0.8736109806742292
Lasso test scores after optimization:
R^2 score: 0.9357184776011483
```

```
In [29]: ridge_pipe = Pipeline([
             ('estimator', Ridge())
         ])
         params_ridge = [{'estimator__alpha': [0.1,0.25,0.5,0.75,1]}]
         ridge_grid = GridSearchCV(estimator=ridge_pipe, param_grid=params_ridge, cv=ShuffleSpli
         ridge_grid.fit(X_train, y_train)
```

```
Out[29]: GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=None, test_size='default',
                train_size=None),
                error_score='raise',
                estimator=Pipeline(memory=None,
              steps=[('estimator', Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=No
            normalize=False, random_state=None, solver='auto', tol=0.001))]),
                fit_params=None, iid=True, n_jobs=1,
                param_grid=[{'estimator__alpha': [0.1, 0.25, 0.5, 0.75, 1]}],
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring=None, verbose=0)
```

```
In [30]: print(ridge_grid.best_params_)
         print(ridge_grid.best_score_)
```

```
{'estimator__alpha': 0.1}
0.9174627803824876
```

```
In [31]: ridge_best = ridge_grid.best_estimator_
         y_pred_ridge_best = ridge_best.predict(X_test)

         print("Ridge test scores before optimization:")
         print("R^2 score: {}".format(r2_score(y_test, y_pred_ridge)))

         print("Ridge test scores after optimization:")
         print("R^2 score: {}".format(r2_score(y_test, y_pred_ridge_best)))
```

```
Ridge test scores before optimization:
R^2 score: 0.8980019834547701
Ridge test scores after optimization:
R^2 score: 0.9406079331347534
```

```
In [32]: elastic_pipe = Pipeline([
             ('estimator', ElasticNet())
         ])
         params_elastic = [{'estimator__alpha': [0.1, 0.25,0.5,0.75,1],
                           'estimator__l1_ratio':[0,0.1,0.25,0.5,0.75,0.9,1]}]
         elastic_grid = GridSearchCV(estimator=elastic_pipe, param_grid=params_elastic, cv=Shuff
         elastic_grid.fit(X_train, y_train)
```

```
Out[32]: GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=None, test_size='default',
                train_size=None),
                error_score='raise',
                estimator=Pipeline(memory=None,
              steps=[('estimator', ElasticNet(alpha=1.0, copy_X=True, fit_intercept=True, l1_rat
               max_iter=1000, normalize=False, positive=False, precompute=False,
               random_state=None, selection='cyclic', tol=0.0001, warm_start=False))]),
                fit_params=None, iid=True, n_jobs=1,
                param_grid=[{'estimator__alpha': [0.1, 0.25, 0.5, 0.75, 1], 'estimator__l1_ratio
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring=None, verbose=0)
```

```
In [33]: print(elastic_grid.best_params_)
         print(elastic_grid.best_score_)
         # l1_ratio tends to be 1, hence, we have a L1 penalty
```

```
{'estimator__alpha': 0.1, 'estimator__l1_ratio': 1}
0.9100179542081691
```

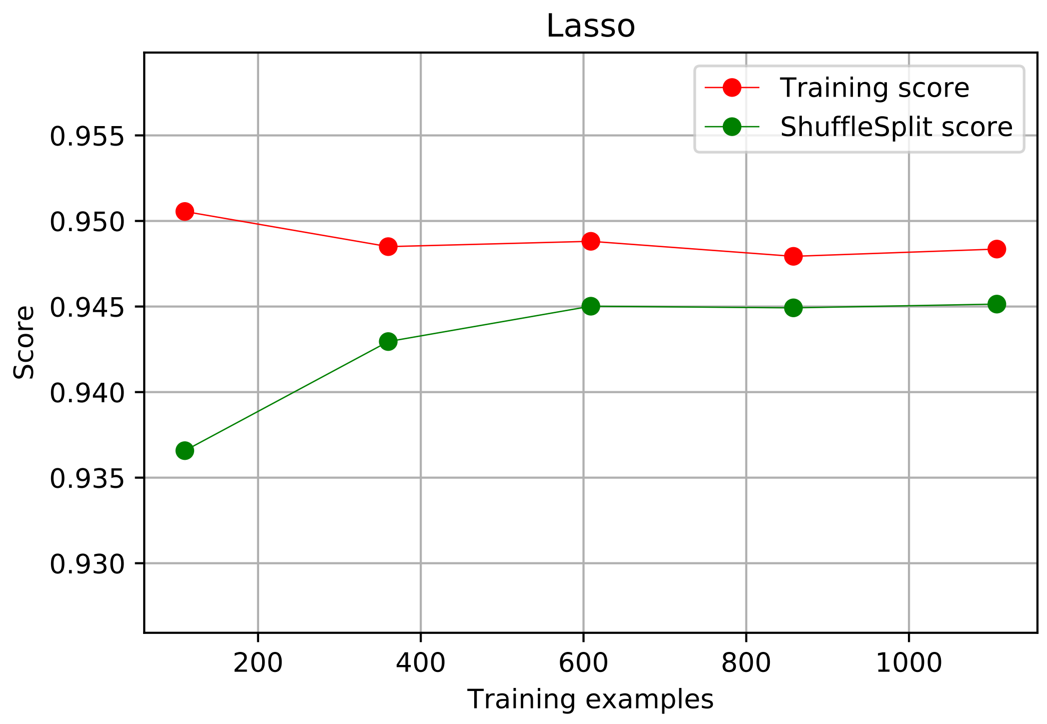
```
In [34]: elastic_best = elastic_grid.best_estimator_
         y_pred_elastic_best = elastic_best.predict(X_test)
```

```
        print("ElasticNet test scores before optimization:")
        print("R^2 score: {}".format(r2_score(y_test, y_pred_elastic)))

        print("ElasticNet test scores after optimization:")
        print("R^2 score: {}".format(r2_score(y_test, y_pred_elastic_best)))
```

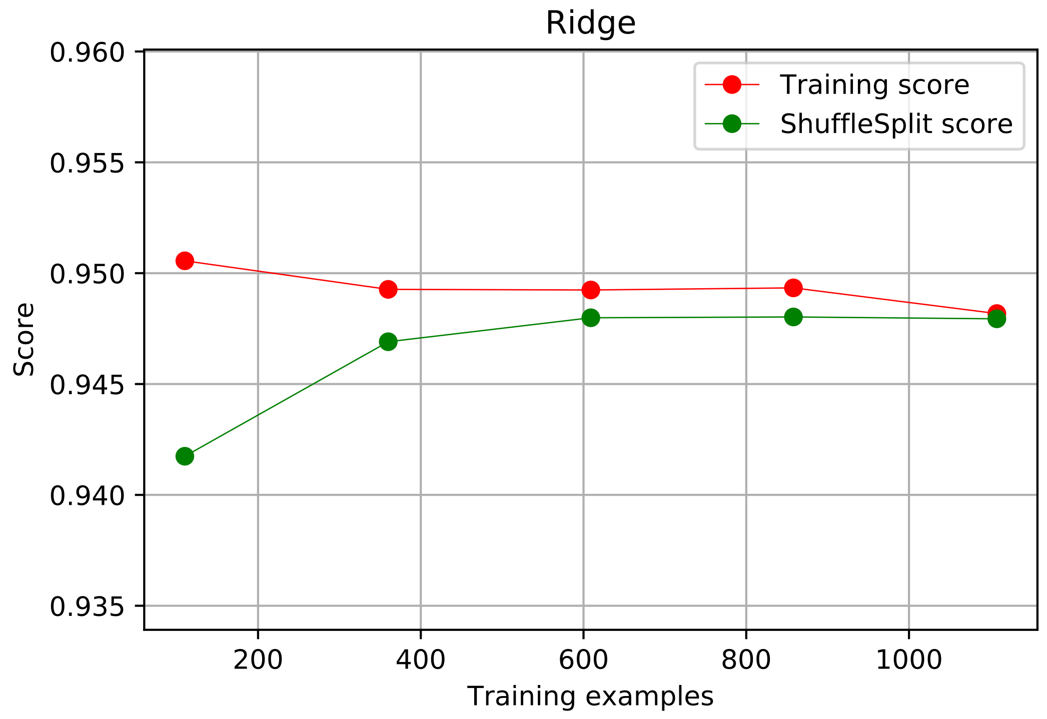
```
ElasticNet test scores before optimization:
R^2 score: 0.8734690094616266
ElasticNet test scores after optimization:
R^2 score: 0.9357184776011483
```

Let's plot the learning curves in of Lasso/Ridge to see whether our optimized value actually optimize the model or if we under/overfit. For simplicity we will choose the highest scoring regularization values manually.

```
In [35]: def plot_learning_curve(alpha_val, title, train_sizes_, train_scores_, test_scores_):
             plt.figure()
             plt.title(title)
         #     if ylim is not None:
         #         plt.ylim(*ylim)
             plt.xlabel("Training examples")
             plt.ylabel("Score")
             train_scores_mean = np.mean(train_scores_, axis=1)
             train_scores_std = np.std(train_scores_, axis=1)
             test_scores_mean = np.mean(test_scores_, axis=1)
             test_scores_std = np.std(test_scores_, axis=1)
             plt.grid()

             plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_me
             plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean
             plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
             plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="ShuffleSplit score"

             plt.legend(loc="best")
             plt.show()

In [36]: train_sizes = np.linspace(.1, 1.0, 5)

In [37]: from sklearn.model_selection import learning_curve

         train_sizes, train_scores, test_scores = learning_curve(Lasso(alpha=0.003), ballframe,
         plot_learning_curve(0.003, "Lasso", train_sizes, train_scores, test_scores)
```



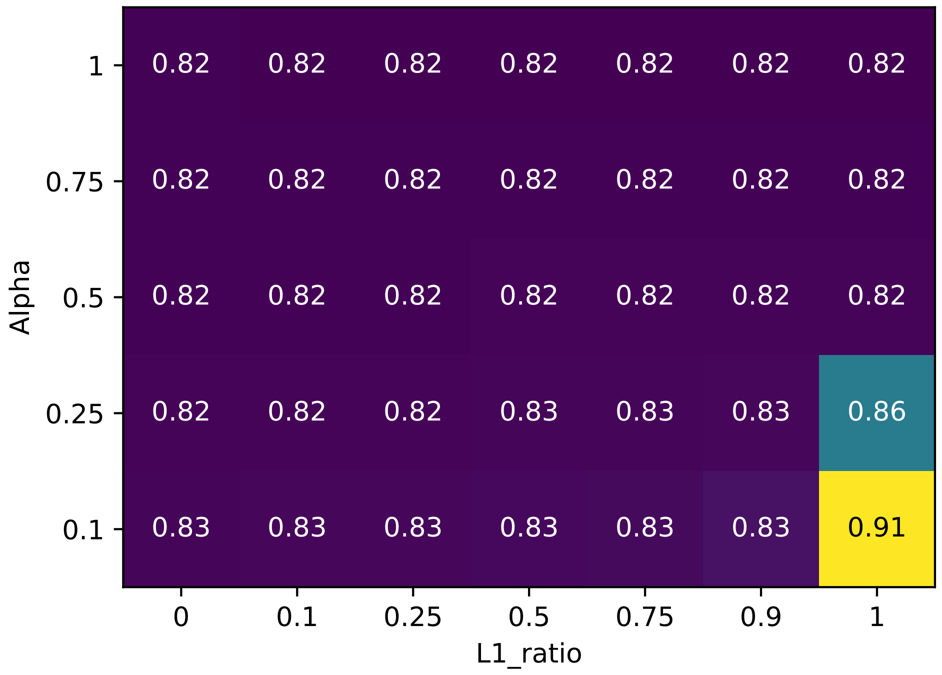
```
In [38]: # ridge
         train_sizes, train_scores, test_scores = learning_curve(Ridge(alpha=0.001), ballframe,
         plot_learning_curve(0.003, "Ridge", train_sizes, train_scores, test_scores)
```



```
In [39]: heat_df = pd.DataFrame(elastic_grid.cv_results_)
         print(heat_df.head(5))
         heat_mean_scores = np.array(heat_df['mean_test_score']).reshape(5,7)

         # plot the mean cross-validation scores
         mglearn.tools.heatmap(heat_mean_scores, xlabel='L1_ratio', xticklabels=params_elastic[0
                               ylabel='Alpha', yticklabels=params_elastic[0]['estimator__alpha']
```

```
   mean_fit_time  mean_score_time  mean_test_score  mean_train_score  \
0       2.32e-02         5.00e-04             0.83              0.86
1       1.20e-03         0.00e+00             0.83              0.86
2       1.76e-03         3.97e-04             0.83              0.86
3       1.10e-03         3.00e-04             0.83              0.86
4       1.60e-03         4.04e-04             0.83              0.86

        ...        std_fit_time std_score_time std_test_score  std_train_score
0       ...            3.68e-03       5.00e-04           0.02         2.03e-03
1       ...            1.83e-03       0.00e+00           0.02         2.03e-03
2       ...            1.89e-03       1.19e-03           0.02         2.03e-03
3       ...            7.00e-04       4.58e-04           0.02         2.02e-03
4       ...            1.96e-03       1.21e-03           0.02         2.00e-03

[5 rows x 32 columns]
```

```
Out[39]: <matplotlib.collections.PolyCollection at 0x21dde326ac8>
```



4 . Visualize the coefficients of the optimized models. Do they agree on which features are important? Compare the results with the feature importances returned by a RandomForest. Do it agree with the linear models? What would look for when scouting for a baseball player?

```
In [40]: elastic_coefs = elastic_grid.best_estimator_.named_steps['estimator'].coef_
         ridge_coefs = ridge_grid.best_estimator_.named_steps['estimator'].coef_
         lasso_coefs = lasso_grid.best_estimator_.named_steps['estimator'].coef_
```

```
In [41]: def normalize_df(df):
             return (df - df.mean()) / (df.max() - df.min())
```

```
In [42]: elastic_coefs_df = pd.DataFrame(elastic_coefs)
         elastic_coefs_df.index = ballframe.columns
         elastic_coefs_df = elastic_coefs_df.sort_values(by=0)

         print('ElasticNet Coefficients:')
         print(elastic_coefs_df)
```

```
ElasticNet Coefficients:
                     0
League       -4.60e+00
OSLG         -3.18e+00
RankSeason   -4.31e-01
Year         -1.51e-01
Team         -5.12e-03
BA            0.00e+00
OOBP         -0.00e+00
RA            5.07e-01
RankPlayoffs  1.08e+00
Playoffs      2.06e+00
G             2.17e+00
W             4.68e+00
OBP           6.30e+02
SLG           9.79e+02
```

```
In [43]: ridge_coefs_df = pd.DataFrame(ridge_coefs)
         ridge_coefs_df.index = ballframe.columns
         ridge_coefs_df = ridge_coefs_df.sort_values(by=0)

         print(ridge_coefs_df)
```

```
                     0
OSLG         -5.76e+00
League       -4.85e+00
RankSeason   -1.20e+00
Year         -1.16e-01
```

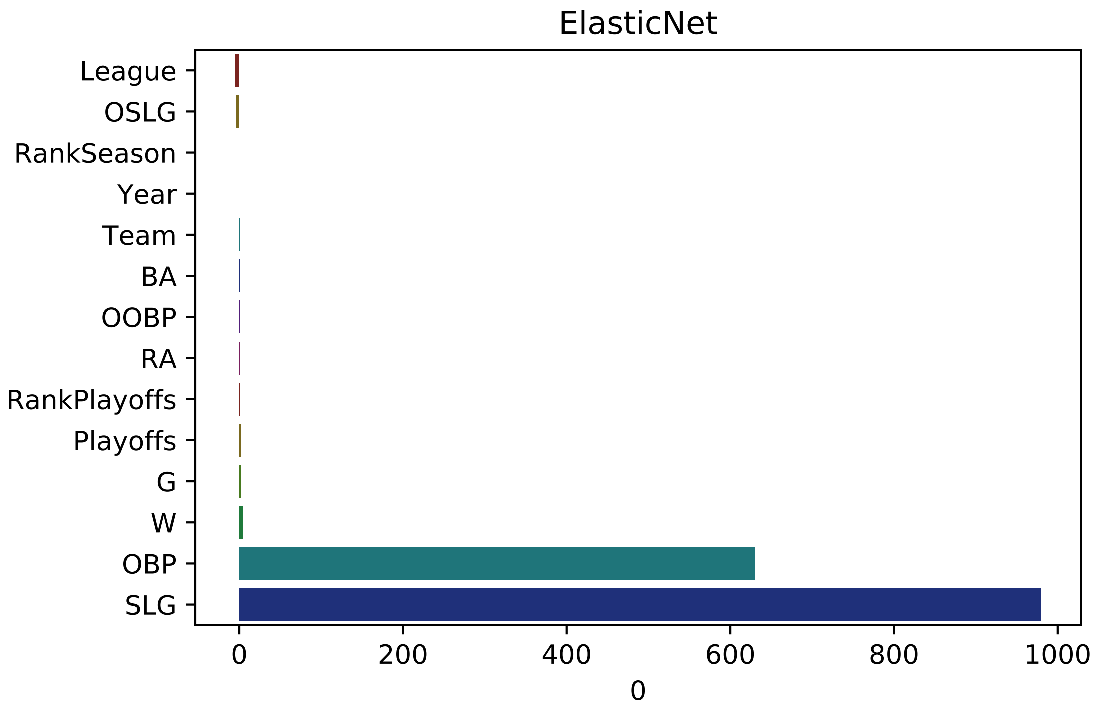
```
Team          -4.59e-03
RA             4.76e-01
OOBP           8.85e-01
RankPlayoffs   1.58e+00
G              2.92e+00
W              4.37e+00
Playoffs       4.37e+00
BA             3.55e+02
OBP            7.43e+02
SLG            9.01e+02
```

```
In [44]: lasso_coefs_df = pd.DataFrame(lasso_coefs)
         lasso_coefs_df.index = ballframe.columns
         lasso_coefs_df = lasso_coefs_df.sort_values(by=0)
         print(lasso_coefs_df)
```

```
                     0
League        -4.60e+00
OSLG          -3.18e+00
RankSeason    -4.31e-01
Year          -1.51e-01
Team          -5.12e-03
BA             0.00e+00
OOBP          -0.00e+00
RA             5.07e-01
RankPlayoffs   1.08e+00
Playoffs       2.06e+00
G              2.17e+00
W              4.68e+00
OBP            6.30e+02
SLG            9.79e+02
```

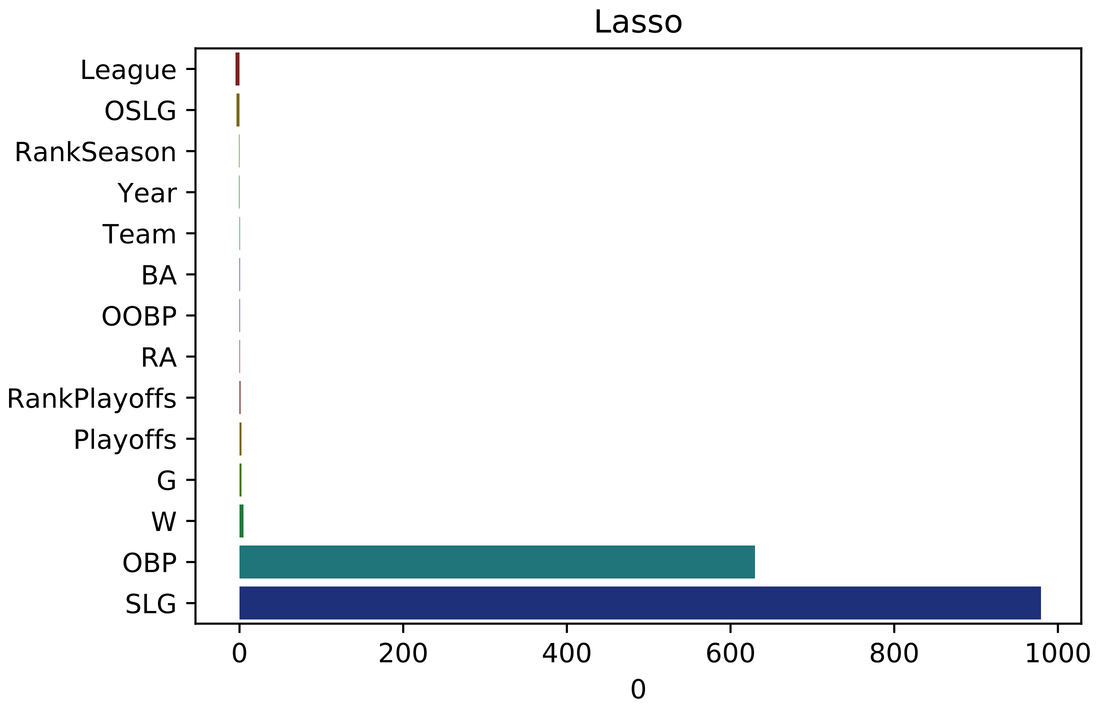
```
In [45]: import seaborn as sns

         sns.barplot(x=elastic_coefs_df[0], y=elastic_coefs_df.index, palette=sns.hls_palette(8,
```

```
Out[45]: Text(0.5,1,'ElasticNet')
```



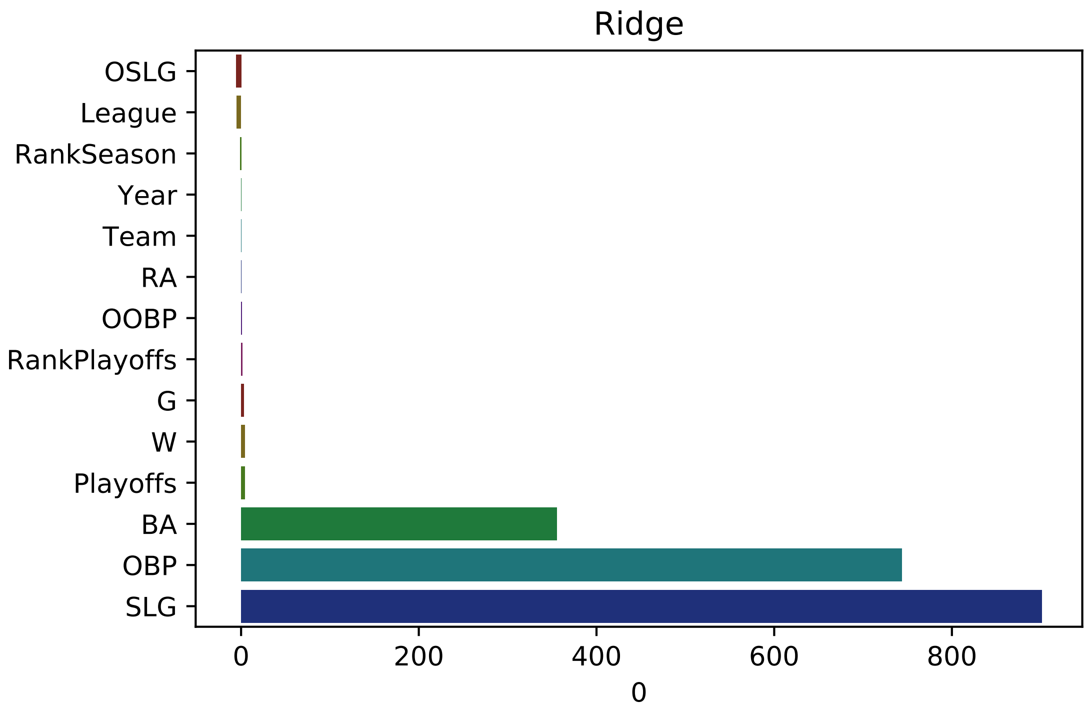
```
In [46]: sns.barplot(x=lasso_coefs_df[0], y=lasso_coefs_df.index, palette=sns.hls_palette(8, l=.
```

```
Out[46]: Text(0.5,1,'Lasso')
```



```
In [47]: sns.barplot(x=ridge_coefs_df[0], y=ridge_coefs_df.index, palette=sns.hls_palette(8, l=.
```

```
Out[47]: Text(0.5,1,'Ridge')
```



From the plots we can clearly see which features are most important to these models. These features are clearly OBP and SLG are dominant in all models while in Ridge BA had a certain influence, too. Lets us have a look at the RandomForest whether it agrees with the regression models.

```
In [48]: from sklearn.ensemble import RandomForestClassifier

         rfc = RandomForestClassifier()
         rfc.fit(X_train, y_train)
```

```
Out[48]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False)
```

```
In [49]: print(rfc.n_features_ )

         print(rfc.feature_importances_)
         print(sorted(zip(map(lambda x: round(x, 3), rfc.feature_importances_), ballframe.column
```

```
14
[0.107 0.031 0.108 0.116 0.117 0.111 0.128 0.111 0.01  0.018 0.015 0.036
 0.048 0.044]
[(0.128, 'SLG'), (0.117, 'W'), (0.116, 'RA'), (0.111, 'OBP'), (0.111, 'BA'), (0.108, 'Year'), (0
```

For the regression models most important features were most definitely SLG and OBP. If we compare the results of a RandomForestClassifier(RFC) with default parameters, we see that the RFC agrees with the most important feature, SLG. RFC, however, regards BA and RA higher. While RA was not present in the regression models, BA was observed to be a more important feature in Ridge. Another notable point in RFC is having surprisingly high importance of Year and Team which could be due to Teams having more impact in certain years such that RFC regards them more important than their actual value.

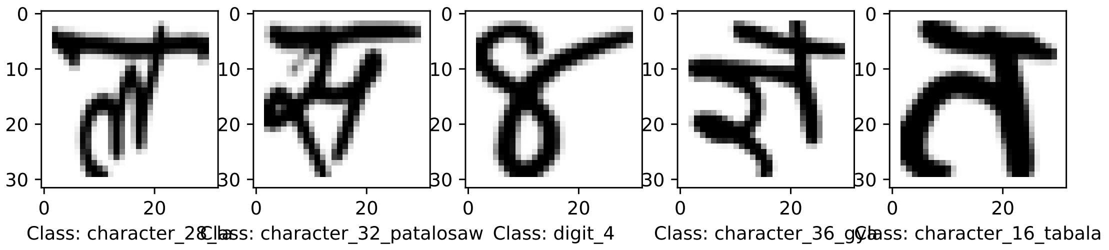
When Scouting for players SLG is clearly the most significant value followed by OBP. Other values to consider are BA and RA.

1.2 Nepalese character recognition (5 points, 1+2+2)

The Devnagari-Script dataset contains 92,000 images (32x32 pixels) of 46 characters from Devanagari script. Your goal is to learn to recognize the right letter given the image.

```
In [90]: devnagari = oml.datasets.get_dataset(40923) # Download Devnagari data
         # Get the predictors X and the labels y
         X, y = devnagari.get_data(target=devnagari.default_target_attribute);
         classes = devnagari.retrieve_class_labels(target_name='character') # This one takes a w
```

```
In [51]: from random import randint
         # Take some random examples, reshape to a 32x32 image and plot
         fig, axes = plt.subplots(1, 5,  figsize=(10, 5))
         for i in range(5):
             n = randint(0,90000)
             axes[i].imshow(X[n].reshape(32, 32), cmap=plt.cm.gray_r)
             axes[i].set_xlabel("Class: %s" % (classes[y[n]]))
         plt.show();
```



```
In [84]: # Training and testing data
         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(
         X, y, test_size=500, train_size=1500, stratify=y) #25-75 split on small dataset, less t

         print(X.shape)

         print("X_train shape: {}".format(X_train.shape))
         print("y_train shape: {}".format(y_train.shape))
         print("X_test shape: {}".format(X_test.shape))
         print("y_test shape: {}".format(y_test.shape))
```

```
(19020, 10)
X_train shape: (1500, 10)
y_train shape: (1500,)
X_test shape: (500, 10)
y_test shape: (500,)
```

We could not fully optimize the models in this assignment. After trying out 10% subsample, all computations were very slow (2h+). Thus, we tried to optimize for a smaller subsample. We are aware that the tune paramaters are most likely not representatives for larger sample size.

1. Evaluate k-Nearest Neighbors, Logistic Regression and RandomForests with their default settings.
   - Take a stratified 10% subsample of the data.
   - Use the default train-test split and predictive accuracy. Is predictive accuracy a good scoring measure for this problem?
   - Try to build the same models on increasingly large samples of the dataset (e.g. 10%, 20%,...). Plot the training time and the predictive performance for each. Stop when the training time becomes prohibitively large (this will be different for different models).

```
In [85]: # k-Nearest Neighbours
         from sklearn.neighbors import KNeighborsClassifier
         knn = KNeighborsClassifier()
         knn.fit(X_train, y_train)

         print("Accuracy on training set: {:.3f}".format(knn.score(X_train, y_train)))
         print("Accuracy on test set: {:.3f}".format(knn.score(X_test, y_test)))

Out[85]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')
```

```
Accuracy on training set: 0.819
Accuracy on test set: 0.768
```

```
In [86]: # Logistic Regression
         from sklearn.linear_model import LogisticRegression
         logreg = LogisticRegression().fit(X_train, y_train)
         print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
         print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))

Training set score: 0.777
Test set score: 0.806
```

```
In [87]: # RandomForests
         from sklearn.ensemble import RandomForestClassifier
         forest = RandomForestClassifier()
         forest.fit(X_train, y_train)

         print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
         print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))

Out[87]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False)

Accuracy on training set: 0.982
Accuracy on test set: 0.844
```

Predictive accuracy my be misleading scoring measure. The model is trained on 1% of the data, on unknown data it performs very bad, less than 0.5. Obviously the training set is not representative for the entire data set.

```
In [88]: import numpy as np
         import matplotlib.pyplot as plt
         from sklearn.model_selection import learning_curve

         def plot_learning_curve(estimator, title, X, y, n_jobs, train_sizes):
             plt.figure()
             plt.title(title)
             plt.xlabel("Training examples")
             plt.ylabel("Score")
             train_sizes, train_scores, test_scores = learning_curve(
                 estimator, X, y, n_jobs=n_jobs, train_sizes=train_sizes)
             train_scores_mean = np.mean(train_scores, axis=1)
             train_scores_std = np.std(train_scores, axis=1)
             test_scores_mean = np.mean(test_scores, axis=1)
             test_scores_std = np.std(test_scores, axis=1)
```

```
        plt.grid()

        plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                         train_scores_mean + train_scores_std, alpha=0.1,
                         color="r")
        plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                         test_scores_mean + test_scores_std, alpha=0.1, color="g")
        plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
                 label="Training score")
        plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
                 label="Cross-validation score")

        plt.legend(loc="best")
        return plt
```

```
In [ ]: from sklearn.model_selection import learning_curve
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.linear_model import LogisticRegression
        from sklearn.ensemble import RandomForestClassifier

        # This cell ran for more than 2 hours on 10% sample. Until then only the first plot was
        estimator=KNeighborsClassifier()
        plt1=plot_learning_curve(estimator, 'KNN', X, y, 2, [0.1, 0.2, 0.3])
        plt1.show()

        estimator=LogisticRegression()
        plt2=plot_learning_curve(estimator, 'Logistic Regression', X, y, 2, [0.1, 0.2, 0.3])
        plt2.show()

        estimator=RandomForestClassifier()
        plt3=plot_learning_curve(estimator, 'Random Forest', X, y, 2, [0.1, 0.2, 0.3])
        plt3.show()
```

2 . Optimize the value for the number of neighbors k (keep $k < 50$) and the number of trees (keep *n_estimators* < 100) on the stratified 10% subsample. - Use 10-fold crossvalidation and plot k and *n_estimators* against the predictive accuracy. Which value of k, *n_estimators* should you pick?

```
In [59]: from sklearn.model_selection import RandomizedSearchCV
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.neighbors import KNeighborsClassifier

         params = {'n_neighbors': [1, 3, 15, 50], 'leaf_size': [30, 40, 50] }
         grid_knn = RandomizedSearchCV(KNeighborsClassifier(n_jobs=2), param_distributions=param
         grid_knn.fit(X_train, y_train)
         print(grid_knn.best_params_)

         params = {'n_estimators': [10, 25, 50, 100] }
         grid_forest = RandomizedSearchCV(RandomForestClassifier(n_jobs=2), param_distributions=
         grid_forest.fit(X_train, y_train)
         print(grid_forest.best_params_)
```

```
Out[59]: RandomizedSearchCV(cv=10, error_score='raise',
                   estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkow
                    metric_params=None, n_jobs=2, n_neighbors=5, p=2,
                    weights='uniform'),
                   fit_params=None, iid=True, n_iter=10, n_jobs=1,
                   param_distributions={'n_neighbors': [1, 3, 15, 50], 'leaf_size': [30, 40, 50]
                   pre_dispatch='2*n_jobs', random_state=None, refit=True,
                   return_train_score='warn', scoring=None, verbose=0)
```

```
{'n_neighbors': 1, 'leaf_size': 30}
```

```
Out[59]: RandomizedSearchCV(cv=10, error_score='raise',
                   estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=2,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False),
                   fit_params=None, iid=True, n_iter=4, n_jobs=1,
                   param_distributions={'n_estimators': [10, 25, 50, 100]},
                   pre_dispatch='2*n_jobs', random_state=None, refit=True,
                   return_train_score='warn', scoring=None, verbose=0)
```

```
{'n_estimators': 100}
```

Initally grid search was used, but it was too slow. From the randomized grid search on 10% sample for knn: 'n_neighbors': 3, 'leaf_size': 50. It took 2h+ to fit, thus, no fine tuning followed. The best parameters for n_neighbors may lie between 1-15.

On smaller sample size, the k = 1, n_estimators=100.

3 . For the RandomForest, optimize both *n_estimators* and *max_features* at the same time on the entire dataset. - Use a nested cross-validation and a random search over the possible values, and measure the accuracy. Explore how fine-grained this grid/random search can be, given your computational resources. What is the optimal performance you find? - Hint: choose a nested cross-validation that is feasible. Don't use too many folds in the outer loop. - Repeat the grid search and visualize the results as a plot (heatmap) $n\_estimators \times max\_features \rightarrow ACC$ with ACC visualized as the color of the data point. Try to make the grid as fine as possible. Interpret the results. Can you explain your observations? What did you learn about tuning RandomForests?

Once again, we could not perform omptimization on entire data set. We optimized on smaller sample set.

```
In [60]: from sklearn.model_selection import cross_val_score
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.model_selection import GridSearchCV
```

```
        params= {'n_estimators': [10, 25, 50, 75, 100], 'max_features': [0.1, 0.2, 0.5, 0.8, 0.
        grid = GridSearchCV(RandomForestClassifier(n_jobs=2), param_grid=params, cv=2)
        grid.fit(X_train, y_train)
        print(grid.best_params_)
        print(grid.best_score_)

        nested_score = cross_val_score(grid, X_train, y_train, cv=2)
        print("Nested cross-validation with train scores: {}".format(nested_score))

Out[60]: GridSearchCV(cv=2, error_score='raise',
                estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='g
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=2,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False),
                fit_params=None, iid=True, n_jobs=1,
                param_grid={'n_estimators': [10, 25, 50, 75, 100], 'max_features': [0.1, 0.2, 0.
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring=None, verbose=0)

{'max_features': 0.1, 'n_estimators': 100}
0.408
Nested cross-validation with train scores: [0.395 0.394]


In [61]: from sklearn.model_selection import RandomizedSearchCV
         from sklearn.ensemble import RandomForestClassifier

         params= {'n_estimators': [10, 25, 50, 75, 100], 'max_features': [0.1, 0.2, 0.5, 0.8, 0.
         random_grid = RandomizedSearchCV(RandomForestClassifier(n_jobs=2), param_distributions
         random_grid.fit(X_train, y_train)
         print(random_grid.best_params_)
         print(random_grid.best_score_)

Out[61]: RandomizedSearchCV(cv=2, error_score='raise',
                   estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=2,
                     oob_score=False, random_state=None, verbose=0,
                     warm_start=False),
                  fit_params=None, iid=True, n_iter=10, n_jobs=1,
                  param_distributions={'n_estimators': [10, 25, 50, 75, 100], 'max_features': [
                  pre_dispatch='2*n_jobs', random_state=None, refit=True,
                  return_train_score='warn', scoring=None, verbose=0)
```

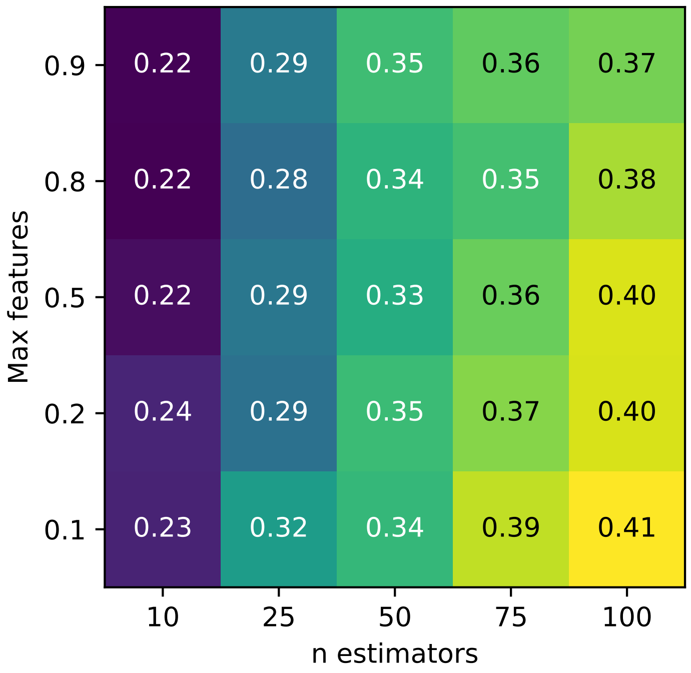
```
{'n_estimators': 100, 'max_features': 0.1}
0.4026666666666667
```

Grid search is most fine-grained. Grid Search has a few downsides: optimizing many hyperparameters creates a combinatorial explosion and a grid must be predefined, hence, you may jump over optimal values. After a number of random search iterations we know more about the performance of hyperparameter settings on the given dataset.

```
In [62]: heat_df = pd.DataFrame(grid.cv_results_)
         print(heat_df.head(5))
         heat_mean_scores = np.array(heat_df['mean_test_score']).reshape(5,5)

         # plot the mean cross-validation scores
         mglearn.tools.heatmap(heat_mean_scores, xlabel='n estimators', xticklabels=params['n_es
                                      ylabel='Max features', yticklabels=params['max_features'], cmap="
```

```
   mean_fit_time  mean_score_time  mean_test_score  mean_train_score  \
0           0.12              0.1             0.23               1.0
1           0.25              0.1             0.32               1.0
2           0.47              0.1             0.34               1.0
3           0.70              0.1             0.39               1.0
4           0.93              0.1             0.41               1.0

        ...        std_fit_time std_score_time std_test_score  std_train_score
0       ...            7.06e-03       6.07e-04       2.57e-02         4.98e-05
1       ...            1.52e-03       1.18e-04       4.67e-03         0.00e+00
2       ...            3.58e-07       1.17e-04       1.22e-02         0.00e+00
3       ...            2.48e-04       9.62e-04       9.94e-03         0.00e+00
4       ...            4.57e-03       2.47e-04       3.84e-04         0.00e+00

[5 rows x 16 columns]
```

```
Out[62]: <matplotlib.collections.PolyCollection at 0x21d80aaa3c8>
```



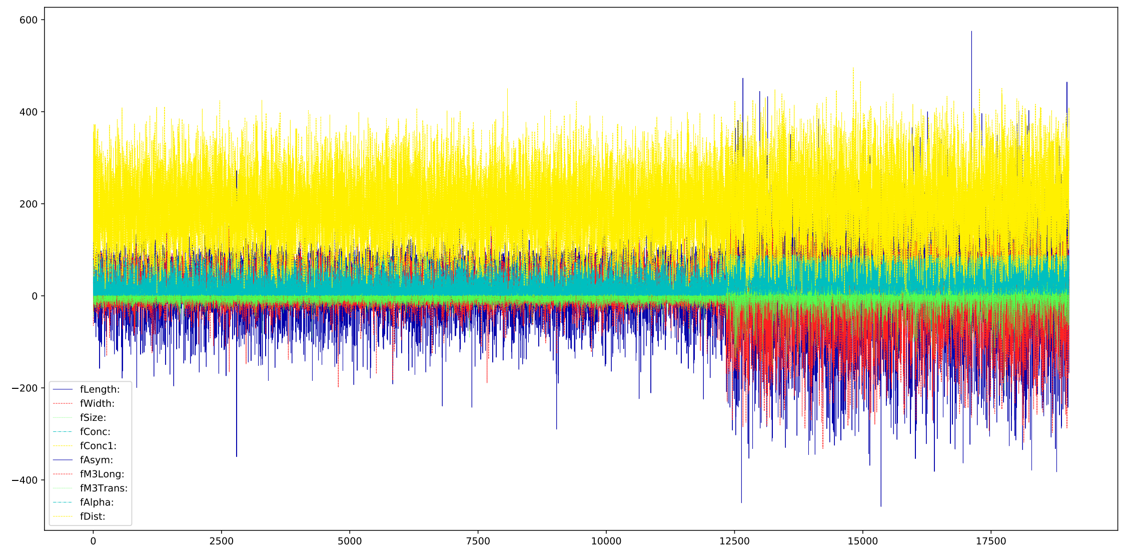
There are 1024 features, but a large amount of them do not carry any information (white pixels). In general, in this dataset, less features give better results, but untul a certain limit. Higher number of extiators is better, but it will start to overfit.

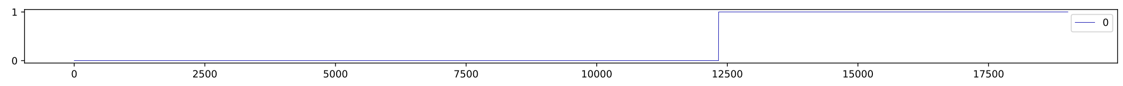
1.3 3. Understanding Ensembles (5 points (3+2))

Do a deeper analysis of how RandomForests and Gradient Boosting reduce their prediction error. We'll use the MAGIC telescope dataset (http://www.openml.org/d/1120). When high-energy particles hit the atmosphere, they produce chain reactions of other particles called 'showers', and you need to detect whether these are caused by gamma rays or cosmic rays.

```
In [63]: # Get the data
         magic_data = oml.datasets.get_dataset(1120) # Download MAGIC Telescope data
         X, y = magic_data.get_data(target=magic_data.default_target_attribute);
```

```
In [64]: # Quick visualization
         X, y, attribute_names = magic_data.get_data(target=magic_data.default_target_attribute,
         magic = pd.DataFrame(X, columns=attribute_names)
         magic.plot(figsize=(20,10))
         # Also plot the target: 1 = gamma, 0 = background
         pd.DataFrame(y).plot(figsize=(20,1));
```





1 . Do a bias-variance analysis of both algorithms. For each, vary the number of trees on a log scale from 1 to 1024, and plot the bias error (squared), variance, and total error (in one plot per algorithm). Interpret the results. Which error is highest for small ensembles, and which reduced most by each algorithm as you use a larger ensemble? When are both algorithms under- or overfitting? Provide a detailed explanation of why random forests and gradient boosting behave this way. - See lecture 3 for an example on how to do the bias-variance decomposition - To save time, you can use a 10% stratified subsample in your initial experiments, but show the plots for the full dataset in your report.

```
In [65]: print(X)
         print(X.dtype)
         print(X.shape)
```

```
[[ 28.797  16.002   2.645 ...  -8.203  40.092  81.883]
 [ 31.604  11.724   2.519 ...  -9.957   6.361 205.261]
 [162.052 136.031   4.061 ... -45.216  76.96   256.788]
 ...
 [ 75.446  47.53    3.448 ...  -9.466  30.299 256.517]
 [120.513  76.902   3.994 ... -63.839  84.687 408.317]
 [187.181  53.001   3.209 ...  31.476  52.731 272.317]]
float32
(19020, 10)
```

```
In [66]: print(y)
         print(y.dtype)

[0 0 0 ... 1 1 1]
int32


In [67]: # print(magic.shape)
         # magic = magic.dropna(axis=0)
         # print(magic.shape)

In [68]: nr_trees = [1,2,4,8,16,32,64,128,256,512,1024]

In [69]: def bias_variance_analysis(clf, X, y, k):
             n_repeat = 100
             shuffle_split = ShuffleSplit(test_size=0.33, n_splits=n_repeat)

             # Store sample predictions
             y_all_pred = [[] for _ in range(len(y))]

             # Train classifier on each bootstrap and score predictions
             for i, (train_index, test_index) in enumerate(shuffle_split.split(X)):
                 # Train and predict
                 clf.fit(X[train_index], y[train_index])
                 y_pred = clf.predict(X[test_index])

                 # Store predictions
                 for i,index in enumerate(test_index):
                     y_all_pred[index].append(y_pred[i])

             # Compute bias, variance, error
             bias_sq = sum([ (1 - x.count(y[i])/len(x))**2 * len(x)/n_repeat
                         for i,x in enumerate(y_all_pred)])
             var = sum([((1 - ((x.count(0)/len(x))**2 + (x.count(1)/len(x))**2))/2) * len(x)/n_r
                        for i,x in enumerate(y_all_pred)])
             error = sum([ (1 - x.count(y[i])/len(x)) * len(x)/n_repeat
                         for i,x in enumerate(y_all_pred)])

             print("Bias squared: %.2f, Variance: %.2f, Total error: %.2f, n_estimators: %.2f" %
             return bias_sq, var, error

In [70]: from sklearn.ensemble import RandomForestClassifier

         rfc_df = pd.DataFrame(columns=['bias_sq', 'var', 'error','n_trees'])

         for i in nr_trees:
             print('begin n_estimators {}'.format(i))
             bias_sq, var, error = bias_variance_analysis(clf=RandomForestClassifier(n_estimator
             rfc_df = rfc_df.append({'bias_sq': bias_sq, 'var':var,'error':error}, ignore_index=
             print('end n_estimators {}'.format(i))

         rfc_df['n_trees']=nr_trees
         print(rfc_df.head())
```

```
begin n_estimators 1
Bias squared: 576.72, Variance: 662.94, Total error: 1239.66, n_estimators: 1.00
end n_estimators 1
begin n_estimators 2
Bias squared: 690.52, Variance: 422.99, Total error: 1113.51, n_estimators: 2.00
end n_estimators 2
begin n_estimators 4
Bias squared: 623.07, Variance: 329.87, Total error: 952.94, n_estimators: 4.00
end n_estimators 4
begin n_estimators 8
Bias squared: 613.04, Variance: 248.14, Total error: 861.18, n_estimators: 8.00
end n_estimators 8
begin n_estimators 16
Bias squared: 616.25, Variance: 188.21, Total error: 804.46, n_estimators: 16.00
end n_estimators 16
begin n_estimators 32
Bias squared: 632.86, Variance: 145.45, Total error: 778.31, n_estimators: 32.00
end n_estimators 32
begin n_estimators 64
Bias squared: 650.32, Variance: 119.03, Total error: 769.35, n_estimators: 64.00
end n_estimators 64
begin n_estimators 128
Bias squared: 655.91, Variance: 101.42, Total error: 757.33, n_estimators: 128.00
end n_estimators 128
begin n_estimators 256
Bias squared: 663.47, Variance: 93.12, Total error: 756.59, n_estimators: 256.00
end n_estimators 256
begin n_estimators 512
Bias squared: 670.48, Variance: 84.91, Total error: 755.39, n_estimators: 512.00
end n_estimators 512
begin n_estimators 1024
Bias squared: 670.79, Variance: 83.43, Total error: 754.22, n_estimators: 1024.00
end n_estimators 1024
   bias_sq     var    error  n_trees
0   576.72  662.94  1239.66        1
1   690.52  422.99  1113.51        2
2   623.07  329.87   952.94        4
3   613.04  248.14   861.18        8
4   616.25  188.21   804.46       16
```

```
In [71]: from sklearn.ensemble import GradientBoostingClassifier
```

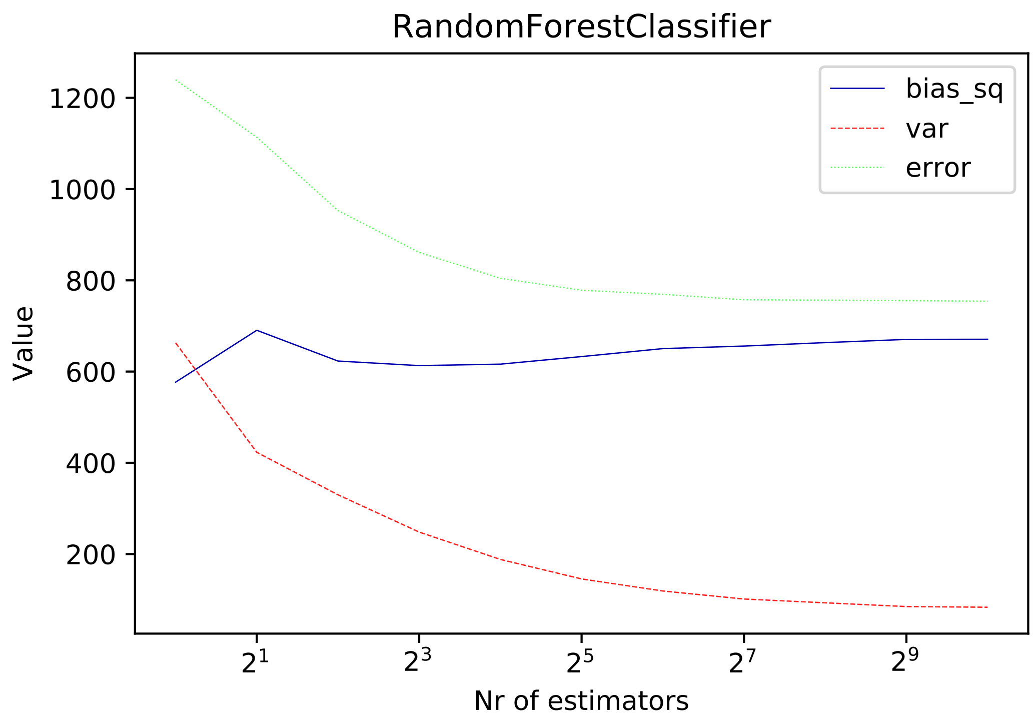
```
gbc_df = pd.DataFrame(columns=['bias_sq', 'var', 'error','n_trees'])

for i in nr_trees:
    print('begin n_estimators {}'.format(i))
    bias_sq, var, error = bias_variance_analysis(clf=GradientBoostingClassifier(n_estim
    gbc_df = gbc_df.append({'bias_sq': bias_sq, 'var':var,'error':error}, ignore_index=
    print('end n_estimators {}'.format(i))

gbc_df['n_trees']=nr_trees
print(gbc_df.head())
```

```
begin n_estimators 1
Bias squared: 2215.40, Variance: 0.00, Total error: 2215.40, n_estimators: 1.00
end n_estimators 1
begin n_estimators 2
Bias squared: 2204.92, Variance: 0.00, Total error: 2204.92, n_estimators: 2.00
end n_estimators 2
begin n_estimators 4
Bias squared: 1351.79, Variance: 62.79, Total error: 1414.58, n_estimators: 4.00
end n_estimators 4
begin n_estimators 8
Bias squared: 1192.14, Variance: 61.32, Total error: 1253.46, n_estimators: 8.00
end n_estimators 8
begin n_estimators 16
Bias squared: 969.74, Variance: 74.59, Total error: 1044.33, n_estimators: 16.00
end n_estimators 16
begin n_estimators 32
Bias squared: 853.90, Variance: 72.52, Total error: 926.42, n_estimators: 32.00
end n_estimators 32
begin n_estimators 64
Bias squared: 766.89, Variance: 71.41, Total error: 838.30, n_estimators: 64.00
end n_estimators 64
begin n_estimators 128
Bias squared: 719.63, Variance: 79.28, Total error: 798.91, n_estimators: 128.00
end n_estimators 128
begin n_estimators 256
Bias squared: 679.97, Variance: 93.89, Total error: 773.86, n_estimators: 256.00
end n_estimators 256
begin n_estimators 512
Bias squared: 640.03, Variance: 113.38, Total error: 753.41, n_estimators: 512.00
end n_estimators 512
begin n_estimators 1024
Bias squared: 610.30, Variance: 136.82, Total error: 747.12, n_estimators: 1024.00
end n_estimators 1024
   bias_sq    var    error  n_trees
0  2215.40   0.00  2215.40        1
1  2204.92   0.00  2204.92        2
2  1351.79  62.79  1414.58        4
```

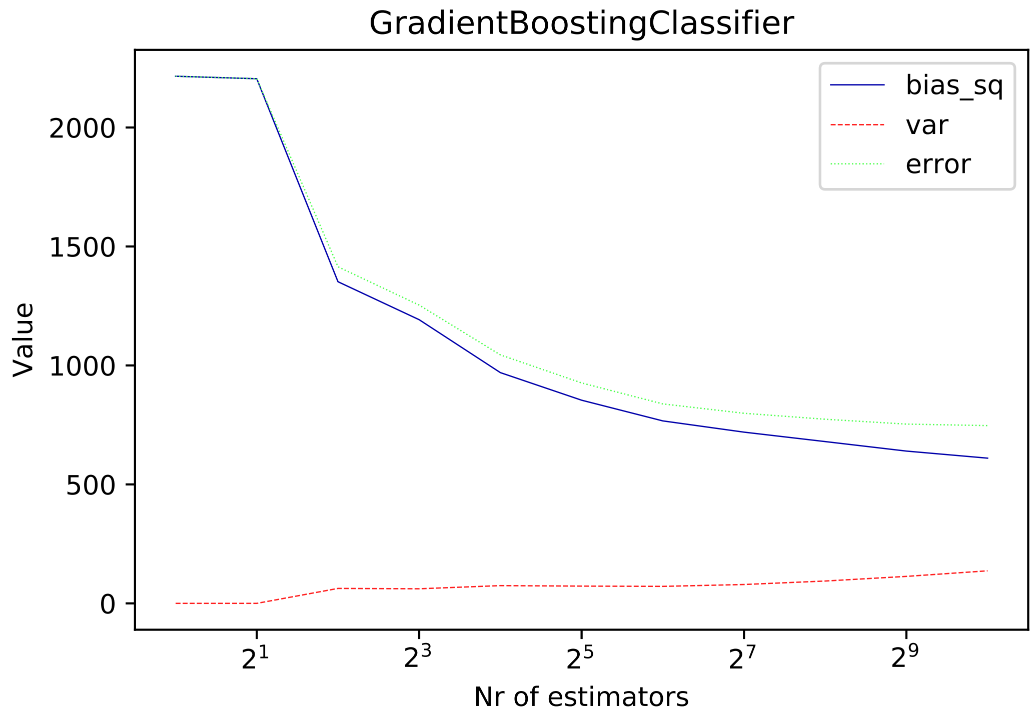
```
3  1192.14  61.32  1253.46        8
4   969.74  74.59  1044.33       16
```

```
In [72]: def plot_vals(df, title):
             fig = plt.figure()

             plt.ylabel('Value')
             plt.xlabel('Nr of estimators')
             plt.title(title)
             plt.xscale('log',basex=2)

             ax = fig.add_subplot(111)
             line1, = ax.plot(df['n_trees'], df['bias_sq'],label='bias_sq', picker=1)
             line2, = ax.plot(df['n_trees'], df['var'], label='var', picker=1)
             line3, = ax.plot(df['n_trees'], df['error'], label='error', picker=1)

             handles, labels = ax.get_legend_handles_labels()
             ax.legend(handles, labels)

             plt.show()
```

```
In [73]: # plot random forest
         plot_vals(rfc_df, 'RandomForestClassifier')
```



```
In [74]: # plot gradient boosting
         plot_vals(gbc_df, 'GradientBoostingClassifier')
```



2 . A *validation curve* can help you understand when a model starts under- or overfitting. It plots both training and test set error as you change certain characteristics of your model, e.g. one or more hyperparameters. Build validation curves for gradient boosting, evaluated using AUROC, by varying the number of iterations between 1 and 500. In addition, use at least two values for the learning rate (e.g. 0.1 and 1), and tree depth (e.g. 1 and 4). This will yield at least 4 curves. Interpret the results and provide a clear explanation for the results. When is the model over- or underfitting? Discuss the effect of the different combinations learning rate and tree depth and provide a clear explanation. - While scikit-learn has a `validation_curve` function, we'll use a modified version (below) that provides a lot more detail and can be used to study more than one hyperparameter. You can use a default train-test split.

```
In [75]: from sklearn.metrics import roc_auc_score
         # Plots validation curves for every classifier in clfs.
         # Also indicates the optimal result by a vertical line
         # Uses 1-AUROC, so lower is better
         def validation_curve(clfs, X_test, y_test, X_train, y_train):
             for n,clf in enumerate(clfs):
                 test_score = np.empty(len(clf.estimators_))
                 train_score = np.empty(len(clf.estimators_))

                 for i, pred in enumerate(clf.staged_decision_function(X_test)):
                     test_score[i] = 1-roc_auc_score(y_test, pred)
```

```
            for i, pred in enumerate(clf.staged_decision_function(X_train)):
                train_score[i] = 1-roc_auc_score(y_train, pred)

            best_iter = np.argmin(test_score)
            learn = clf.get_params()['learning_rate']
            depth = clf.get_params()['max_depth']
            test_line = plt.plot(test_score,
                                 label='learn=%.1f depth=%i (%.2f)'%(learn,depth,
                                                                    test_score[best_iter]))

            colour = test_line[-1].get_color()
            plt.plot(train_score, '--', color=colour)

            plt.xlabel("Number of boosting iterations")
            plt.ylabel("1 - area under ROC")
            plt.axvline(x=best_iter, color=colour)

        plt.legend(loc='best')
```

In [76]:
```
X_train, X_test, y_train, y_test = train_test_split(magic, y, random_state=0)
```

In [77]:
```
# clf0 = GradientBoostingClassifier(learning_rate=10, max_depth=20)
# clf0.fit(X_train, y_train)

clf1 = GradientBoostingClassifier(learning_rate=0.1, max_depth=1)
clf1.fit(X_train, y_train)

clf2 = GradientBoostingClassifier(learning_rate=1, max_depth=1)
clf2.fit(X_train, y_train)

clf3 = GradientBoostingClassifier(learning_rate=0.1, max_depth=4)
clf3.fit(X_train, y_train)

clf4 = GradientBoostingClassifier(learning_rate=1, max_depth=4)
clf4.fit(X_train, y_train)

clfs = [clf1,clf2,clf3,clf4]
```

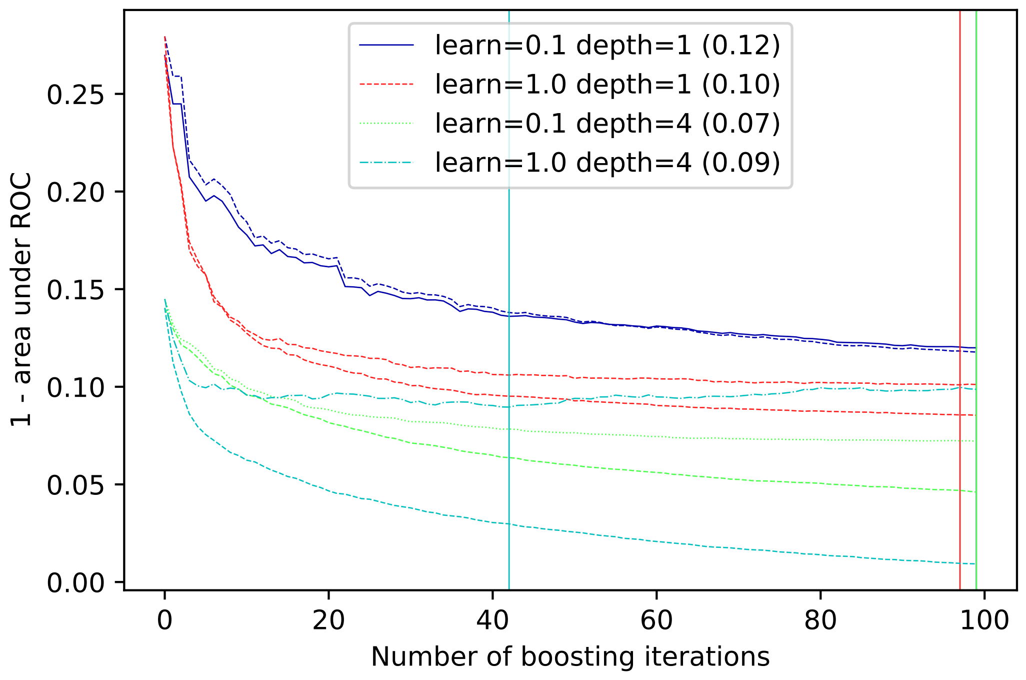
Out[77]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=1,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=None, subsample=1.0, verbose=0,
              warm_start=False)
```

Out[77]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=1, loss='deviance', max_depth=1,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=100,
                     presort='auto', random_state=None, subsample=1.0, verbose=0,
                     warm_start=False)
```

```
Out[77]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                     learning_rate=0.1, loss='deviance', max_depth=4,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=100,
                     presort='auto', random_state=None, subsample=1.0, verbose=0,
                     warm_start=False)
```

```
Out[77]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                     learning_rate=1, loss='deviance', max_depth=4,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=100,
                     presort='auto', random_state=None, subsample=1.0, verbose=0,
                     warm_start=False)
```

```
In [78]: validation_curve(clfs, X_test, y_test, X_train, y_train)
```



We run the classifier in all four cases with 100 estimators to be able to compare its results. We distinct between four different curves:

learning_rate=0.1, max_depth=1(blue line): This model is underfitting because we set the learning rate to 0.1 but also have 'only' 100 estimators, therefore, the model may not be complex. Additionally, max_depth is equal to 1 which reduces the complexitivity even further. In the plot we can also observe that the best classifier is found at 100 estimators.

learning_rate=1, max_depth=1(red line): The red line behaves similar to the blue line. We can observe that at the same max_depth as the blue line, the red line has the best score at slightly less estimators. Nevertheless, the classifier may still underfit due to the low max_depth and reduced complexity.

learning_rate=0.1, max_depth=4(green line): This model is faily balanced. The learning rate is set to 0.1 which would require the model to have a more complex structure. The complex structure is given by the max_depth of 4.

learning_rate=1, max_depth=4(light blue): The last model will overfit due to high learning rate and high max_depth. When the learning rate is set to 1, less trees are need to get comlex models. Max_depth on the other hand is set to 4 resulting in deeper tree structures. Another point that shows the overfitting of the model is that it converges at roughly 40 iterations.