Assignment 1

March 5, 2018

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

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 = "Irene Septyo Rini Panuju"
NAME_STUDENT_2 = "Rumjana Romanova"
```

In [2]:
```
%matplotlib inline
from preamble import *
import seaborn as sns
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 [3]:
```
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, r
# Describe the data with pandas, just to get an overview
ballframe = pd.DataFrame(X, columns=attribute_names)
ballframe.describe()
type(X)
type(ballframe)
```

Out[3]:

	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]

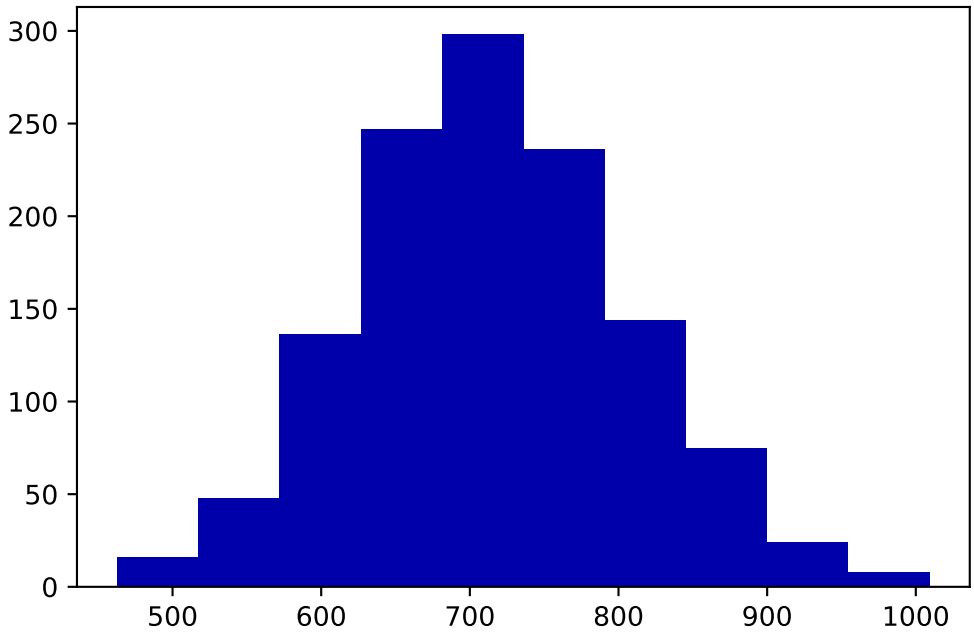
Out[3]: numpy.ndarray

Out[3]: pandas.core.frame.DataFrame

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)

In [6]:
```
#histogram for target
h = plt.hist(y)
```

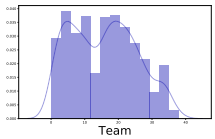



```
In [4]: print("Targets:\n{}".format(moneyball.default_target_attribute))

        plt.rcParams['figure.figsize'] = (10,100)
        fig, axs =plt.subplots(X.shape[1],1)
        ballframe.dtypes

        for i in range(0, X.shape[1]):
            # check data types
            type(X[1][i])

            X_cl = X[:, i][np.logical_not(np.isnan(X[:,i]))]
            # shape to see which ones contain NA values
            X_cl.shape
            sns.distplot(X_cl,ax = axs[i]).set_xlabel(attribute_names[i],fontsize=34)
        fig.show()

Targets:
RS


Out[4]: Team            float32
        League          float32
        Year            float32
        RA              float32
        W               float32
        OBP             float32
```

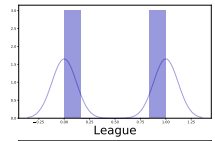
```
SLG             float32
BA              float32
Playoffs        float32
RankSeason      float32
RankPlayoffs    float32
G               float32
OOBP            float32
OSLG            float32
dtype: object
```

```
Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'Team')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'League')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'Year')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'RA')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'W')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'OBP')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'SLG')
```

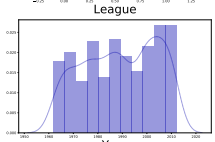
```
Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'BA')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'Playoffs')

Out[4]: numpy.float32

Out[4]: (244,)

Out[4]: Text(0.5,0,'RankSeason')

Out[4]: numpy.float32

Out[4]: (244,)

Out[4]: Text(0.5,0,'RankPlayoffs')

Out[4]: numpy.float32

Out[4]: (1232,)

Out[4]: Text(0.5,0,'G')

Out[4]: numpy.float32

Out[4]: (420,)

Out[4]: Text(0.5,0,'OOBP')

Out[4]: numpy.float32

Out[4]: (420,)

Out[4]: Text(0.5,0,'OSLG')
```



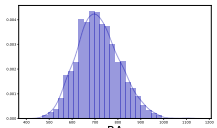
Team



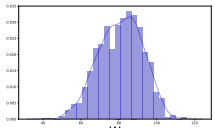
League



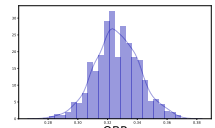
Year



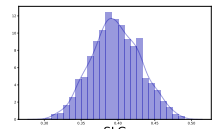
RA



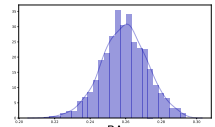
W



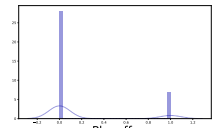
OBP



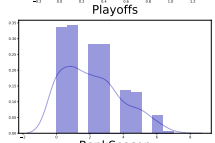
SLG



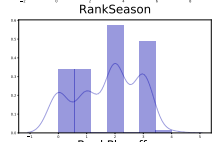
BA



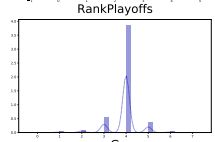
Playoffs



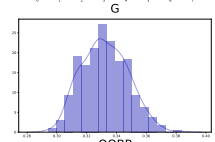
RankSeason



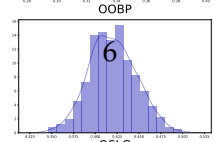
RankPlayoffs



G

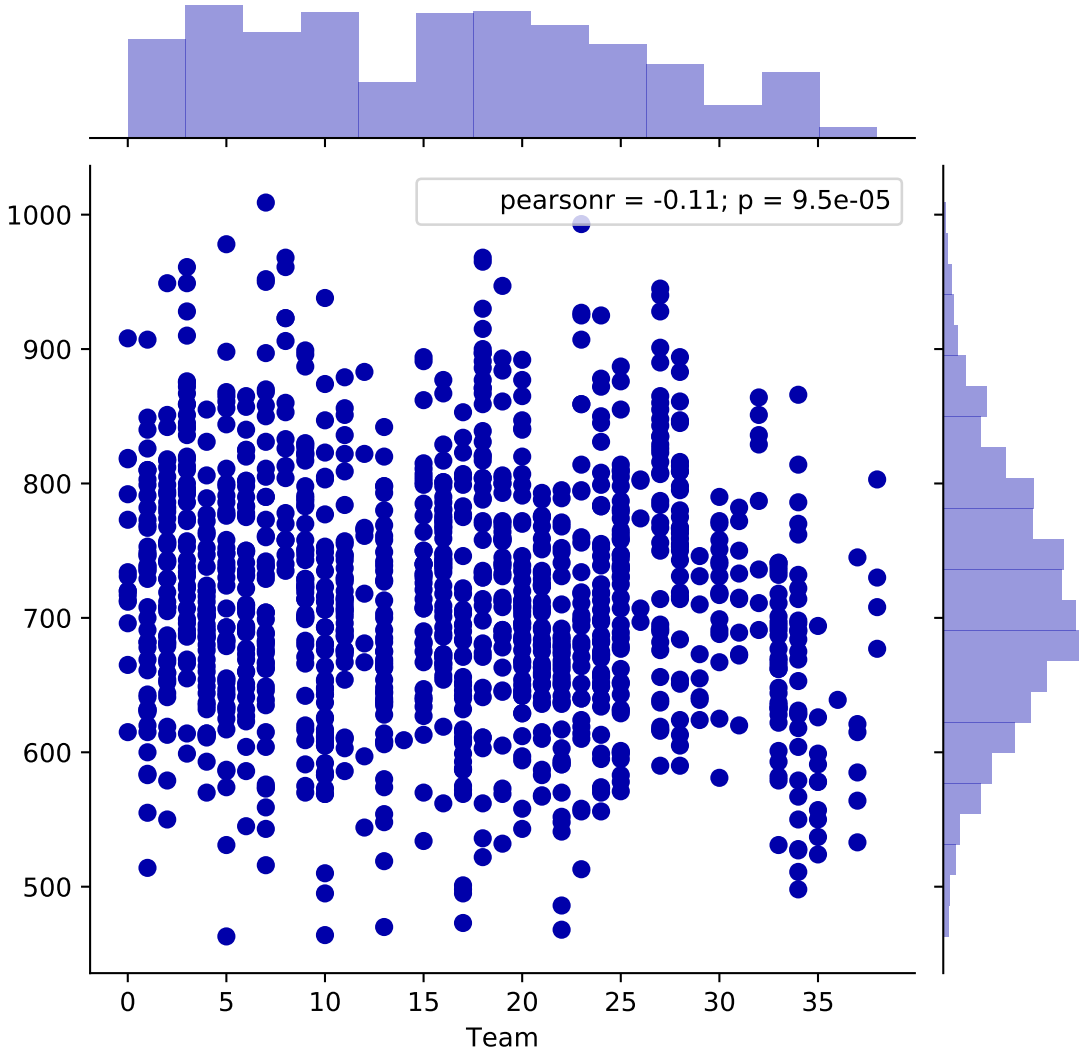


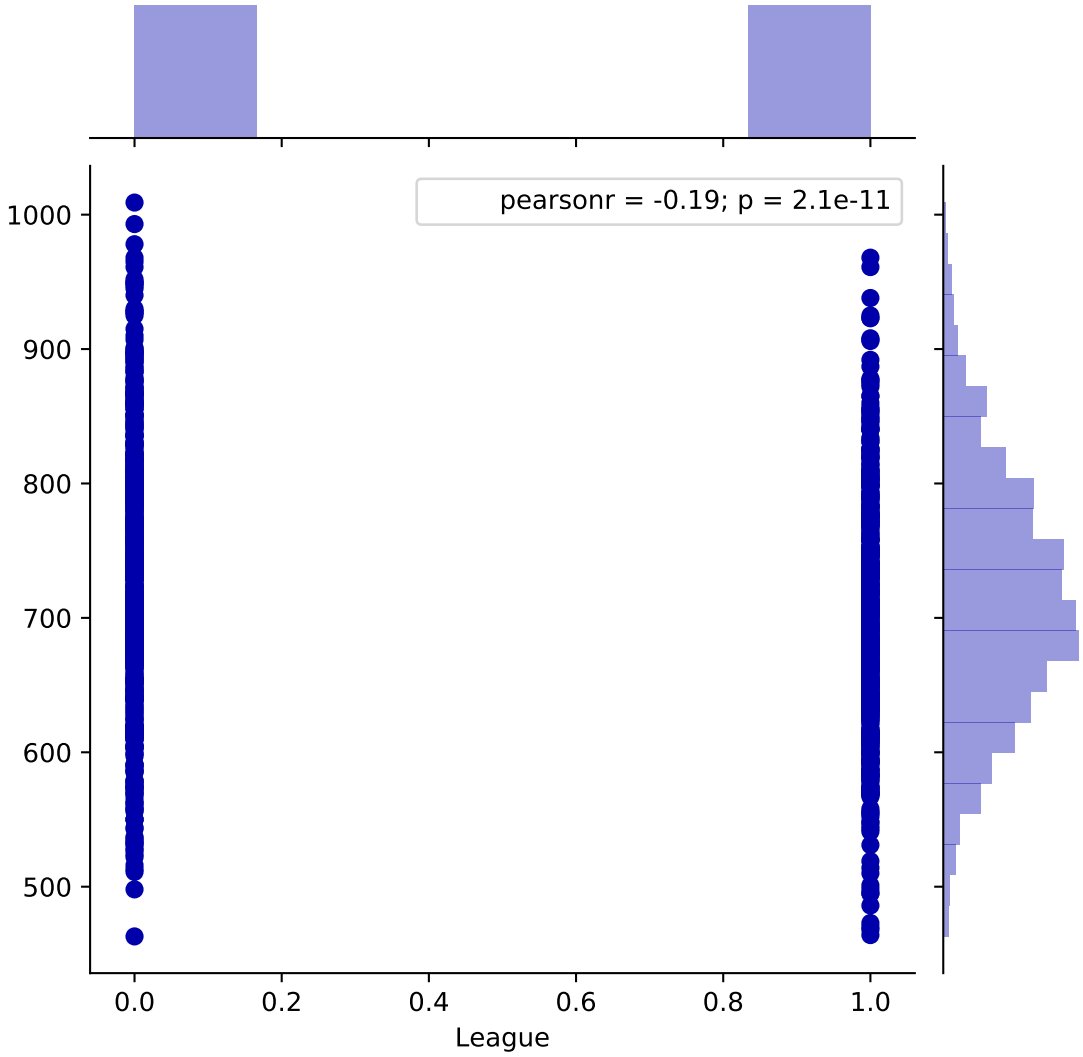
OOBP

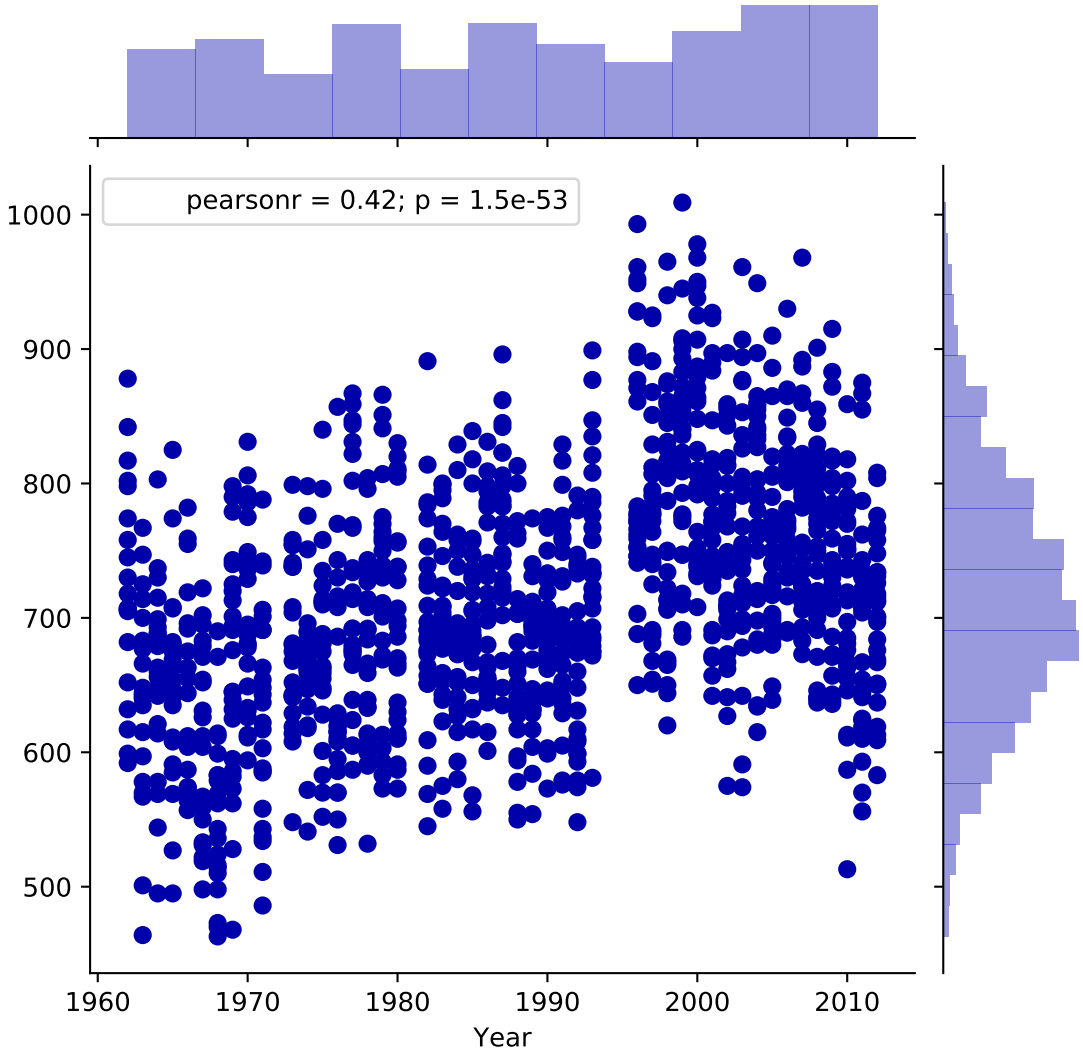


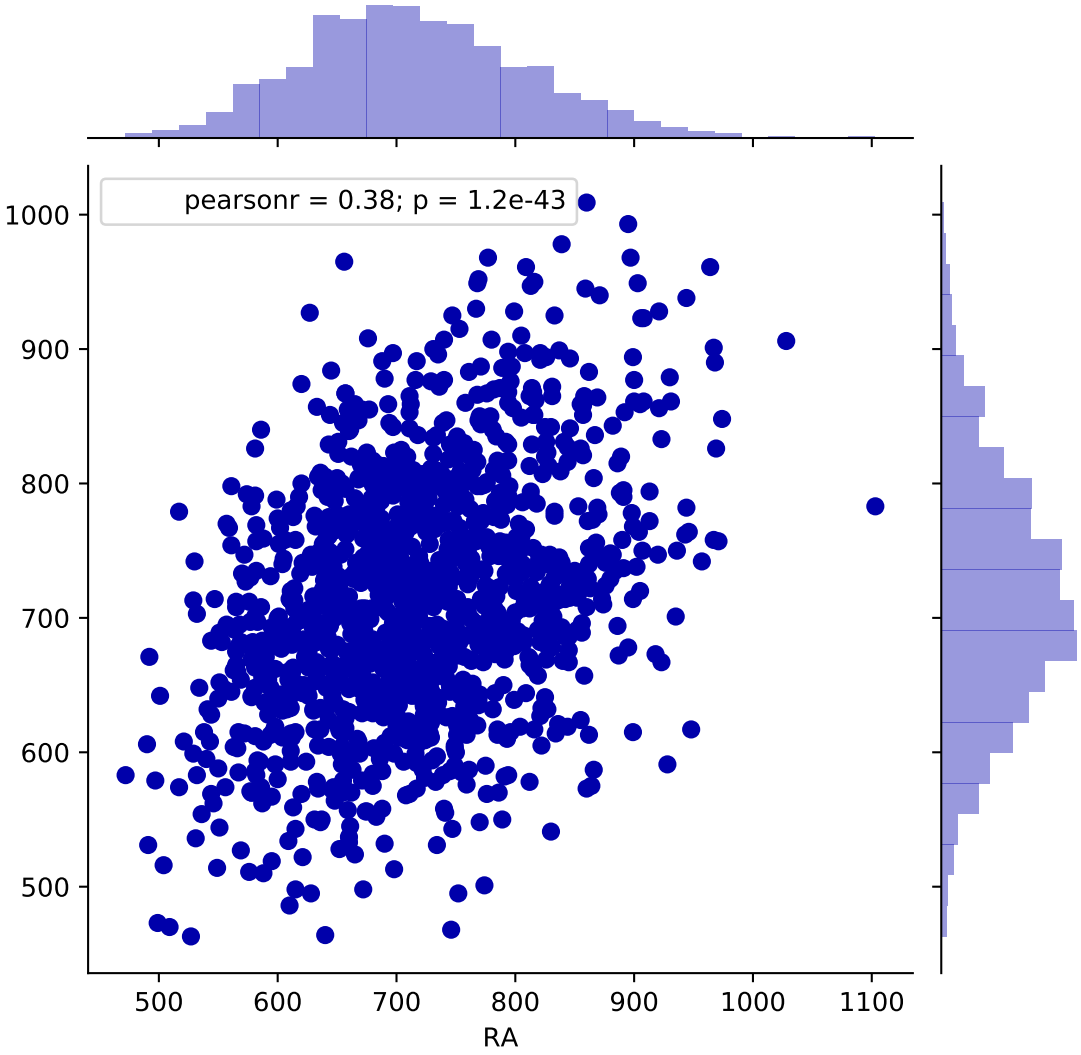
OSLG

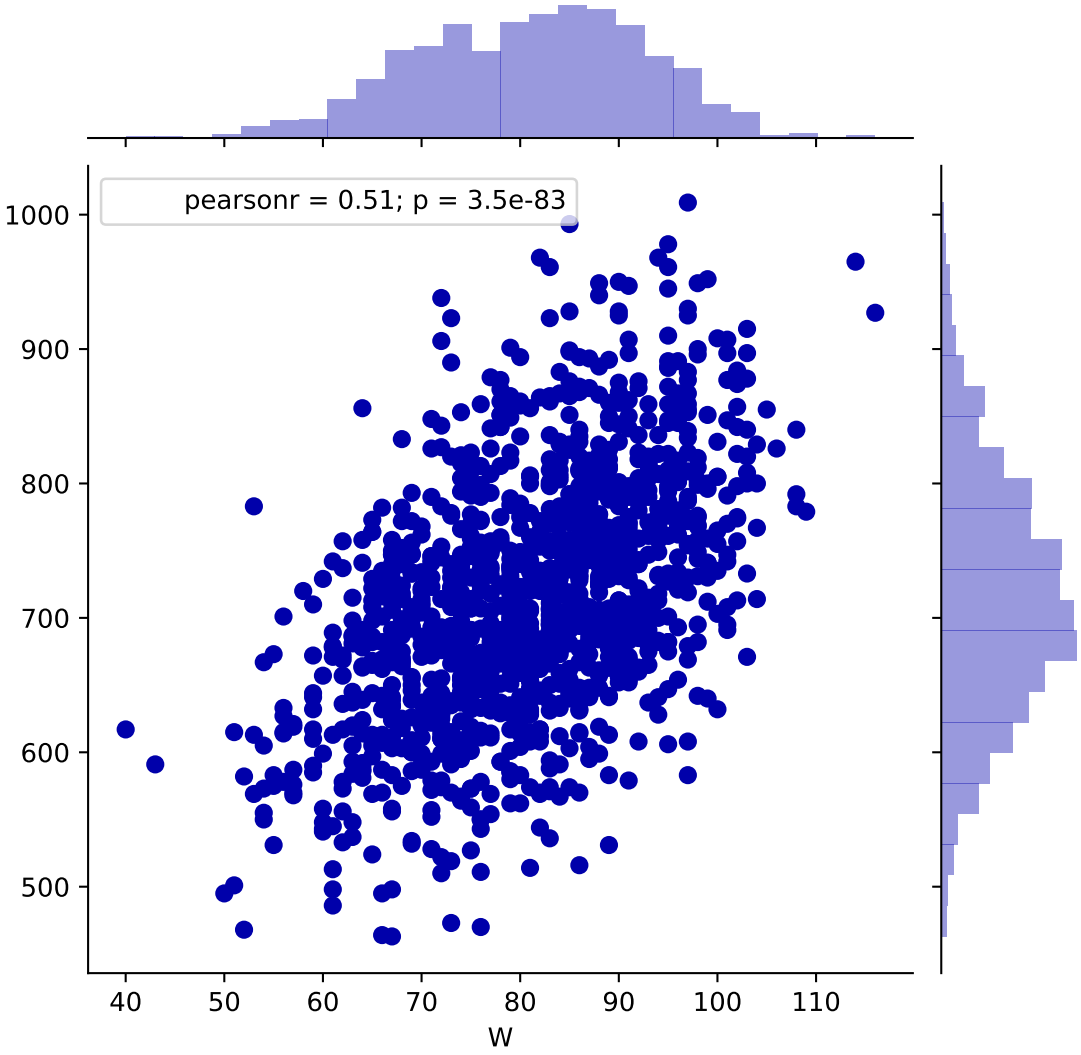
```
In [5]: plt.rcParams['figure.figsize'] = (10,10)

        ballframe.index

        bf_new = ballframe.dropna(axis=1)
        bf_new.columns
        for col in bf_new.columns:
            sns.jointplot(bf_new[col], y)

Out[5]: RangeIndex(start=0, stop=1232, step=1)

Out[5]: Index(['Team', 'League', 'Year', 'RA', 'W', 'OBP', 'SLG', 'BA', 'Playoffs',
               'G'],
              dtype='object')

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a845923ef0>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846cf05c0>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846da6cf8>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846e53ac8>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846f34c88>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846ffbd30>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a8470c3f28>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a846e539b0>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a84717dfd0>

Out[5]: <seaborn.axisgrid.JointGrid at 0x2a847327be0>
```

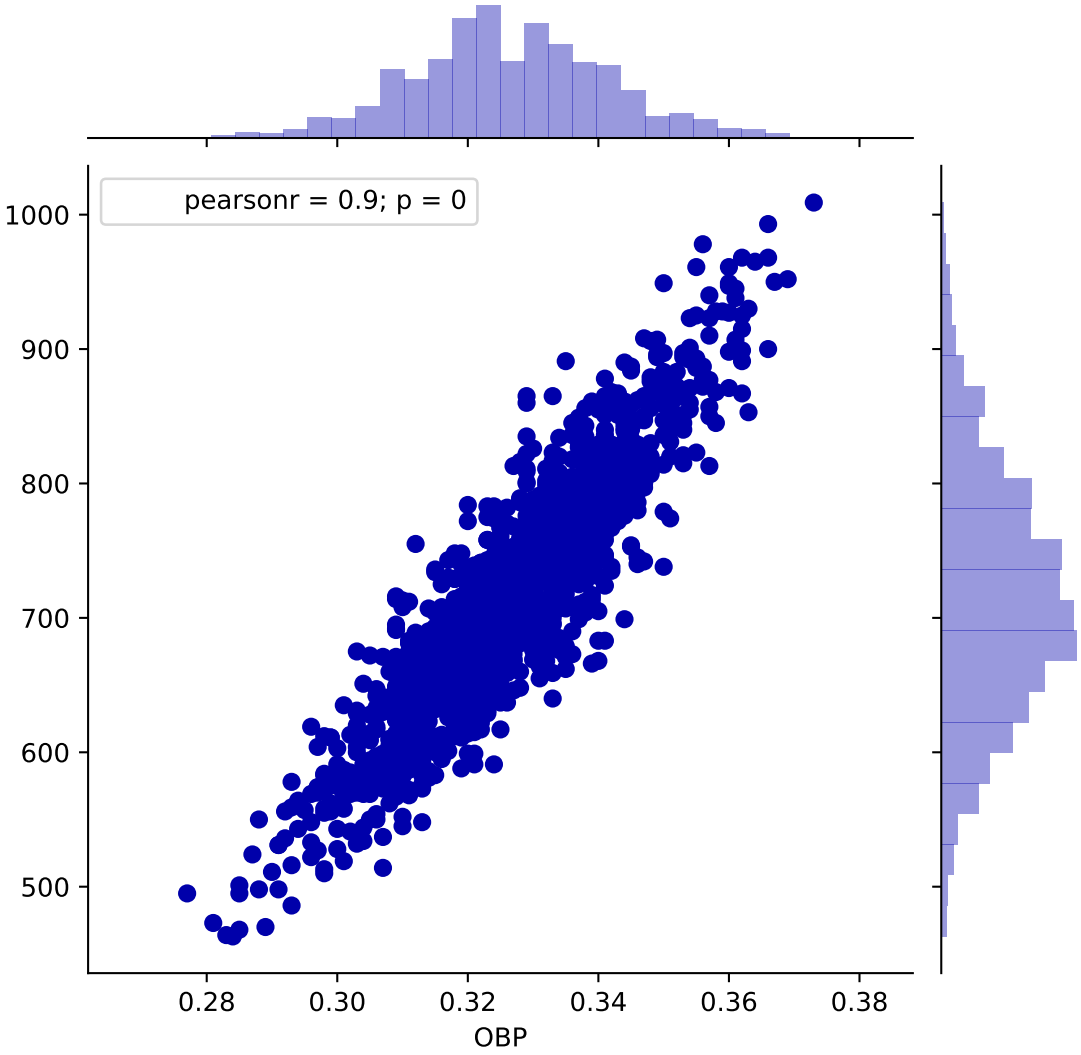

pearsonr = -0.11; p = 9.5e-05
1000
900
800
700
600
500
0
5
10
15
20
25
30
35
Team

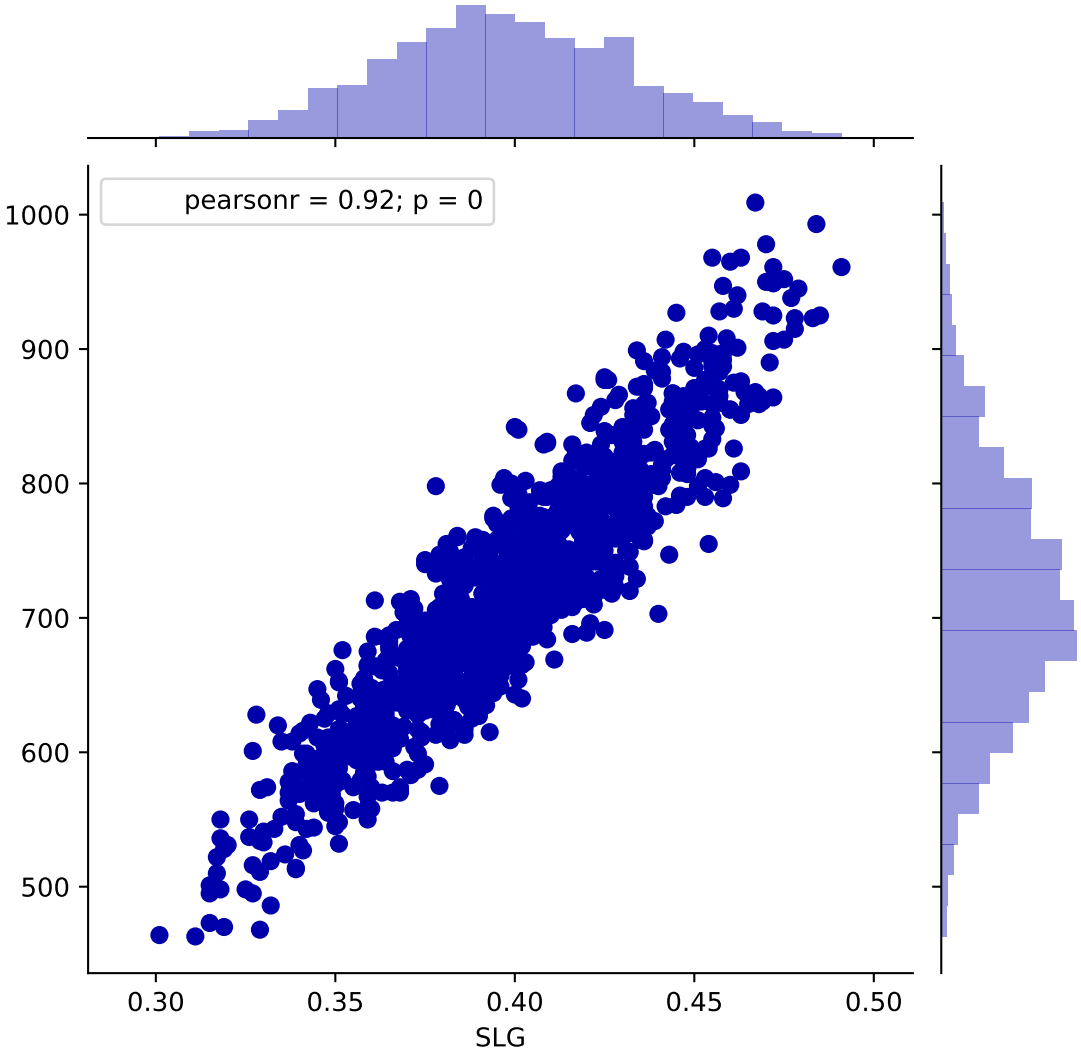

pearsonr = -0.19; p = 2.1e-11
1000
900
800
700
600
500
0.0
0.2
0.4
0.6
0.8
1.0
League

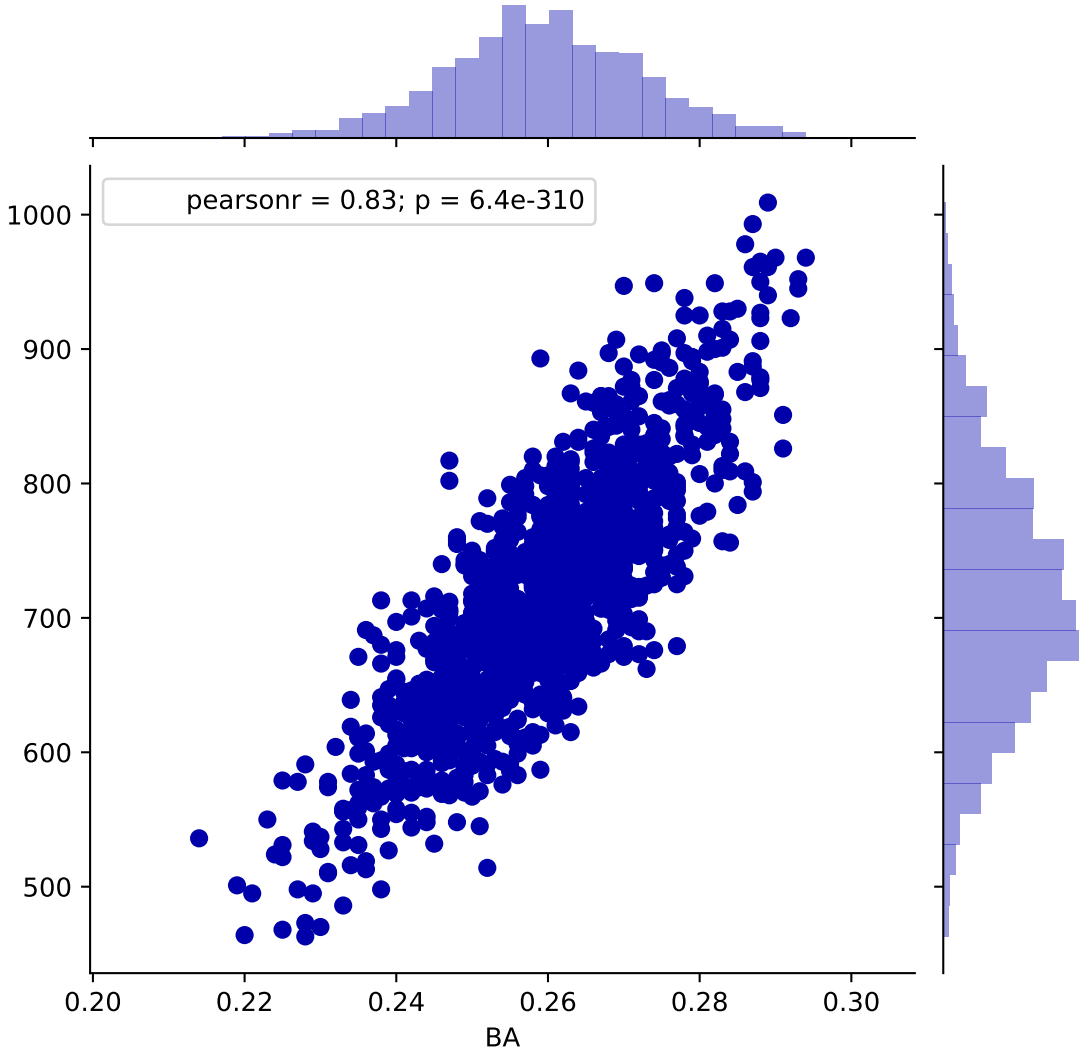

pearsonr = 0.42; p = 1.5e-53
1000
900
800
700
600
500
1960
1970
1980
1990
2000
2010
Year

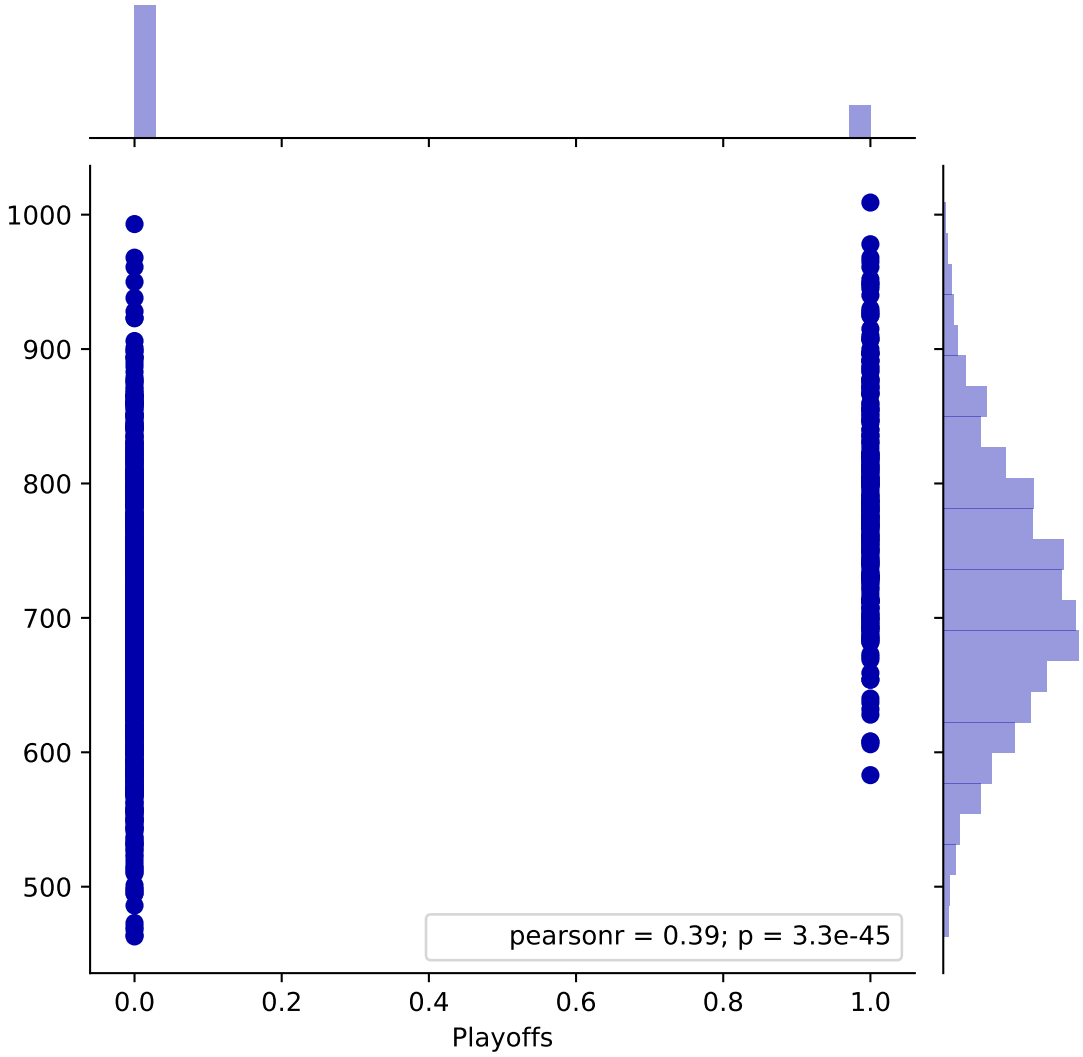

pearsonr = 0.38; p = 1.2e-43
1000
900
800
700
600
500
500
600
700
800
900
1000
1100
RA

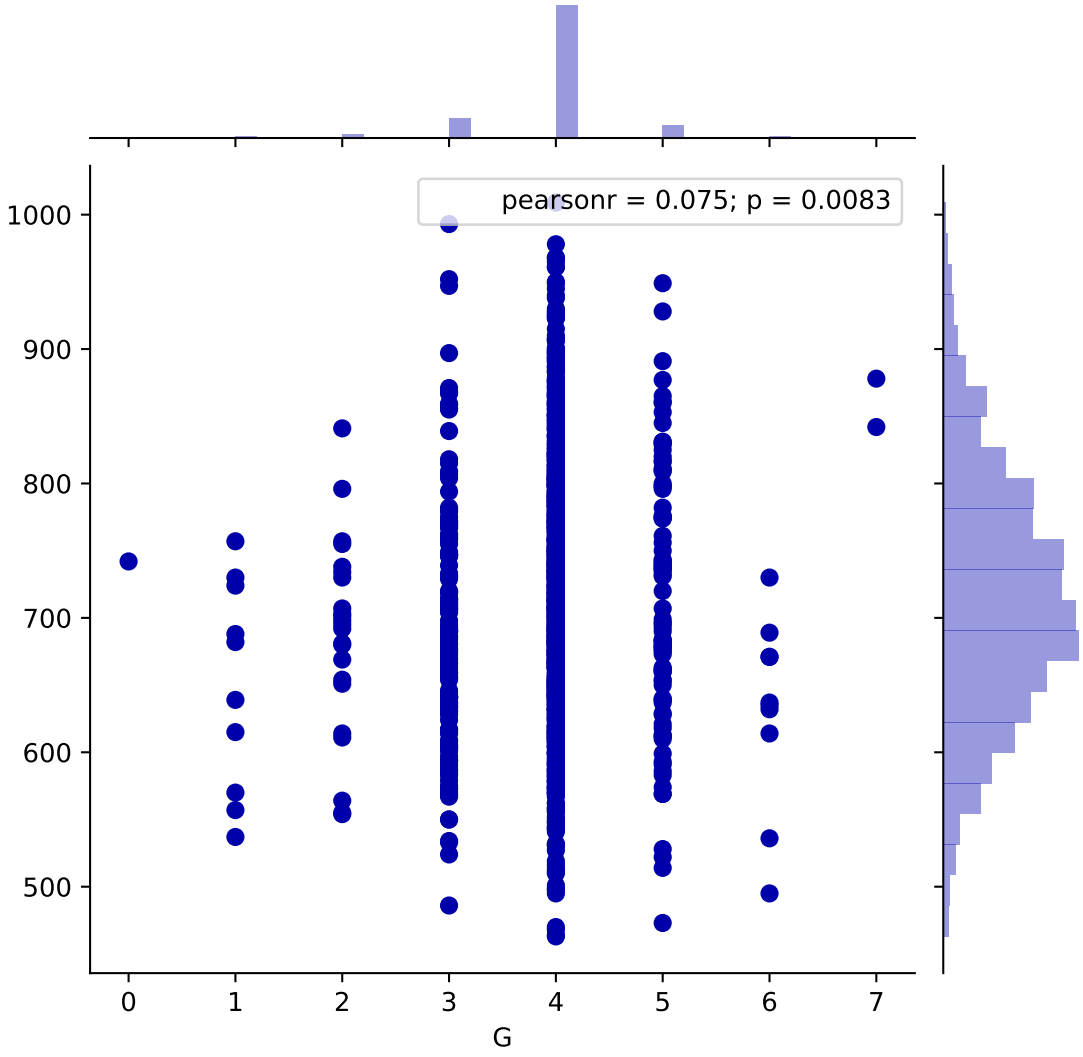

pearsonr = 0.51; p = 3.5e-83
1000
900
800
700
600
500
40
50
60
70
80
90
100
110
W


pearsonr = 0.9; p = 0
1000
900
800
700
600
500
0.28
0.30
0.32
0.34
0.36
0.38
OBP


pearsonr = 0.92; p = 0
1000
900
800
700
600
500
0.30
0.35
0.40
0.45
0.50
SLG


pearsonr = 0.83; p = 6.4e-310
1000
900
800
700
600
500
0.20
0.22
0.24
0.26
0.28
0.30
BA


1000
900
800
700
600
500
0.0
0.2
0.4
0.6
0.8
1.0
Playoffs
pearsonr = 0.39; p = 3.3e-45



From the scatter plots above, we can see that variables RA, W, OBP, SLG, and BA have some linear correlation with Targets. Therefore, it's probably useful to pay close attention to these variables. By inspecting the columns and their NA values, it becomes apparent that RankSeason and RankPlayoffs contain only 244 observations, where actual value is present. Similarly, OOBP and OSLG contain only 420 observations which are not NA.
In this regard, we think that attributes containing NA values might require special treatment, as NA values self cannot be used in the models. A possibility is to remove all rows with missing data, but that will results with way too few observations ~200 from data set of 1200. So due to the small resultign data set, this was not chosen as an option. Another possibility is filling in the NA values with the average values of the attribute, but as we think that such an approach may skew too much the data set, then it was decided simply to drop these 4 attributes for the future when models are created.

Furthermore, datatypes of all attributes are checked in order to ensure that no string data in present, but it is encoded. Later, we found out as well that 'get_data' from OpenML deals automatically with the encoding of nominal features.

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 [6]: # Linear Regression
        from sklearn.linear_model import LinearRegression
        from sklearn.linear_model import LinearRegression
        from sklearn.model_selection import cross_val_score
        from sklearn.pipeline import Pipeline
        from sklearn.preprocessing import StandardScaler
        from sklearn.model_selection import train_test_split

        print("Results of Linear Regression")
        X_train, X_test, y_train, y_test = train_test_split(bf_new, y, random_state=0)

        # normal linear regression
        lr = LinearRegression().fit(X_train, y_train)
        print("Training set score (R^2): {:.2f}".format(lr.score(X_train, y_train)))
        print("Test set score (R^2): {:.2f}".format(lr.score(X_test, y_test)))

        # lr on scaled data - with Standard Scaler
        scaler = StandardScaler(with_mean = False)
        X_train_scaled = scaler.fit_transform(X_train)
        X_test_scaled = scaler.transform(X_test)
        lr_sc = LinearRegression().fit(X_train_scaled, y_train)
        print("Training set score on scaled data (R^2): {:.2f}".format(lr_sc.score(X_train_scal
        print("Test set score on scaled data (R^2): {:.2f}".format(lr_sc.score(X_test_scaled, y

        # lr with CV
        lin_reg = LinearRegression()
        scores_lr_cv = cross_val_score(lin_reg, bf_new, y)
        print("Cross-validation scores: {}".format(scores_lr_cv))
        print("Average Cross-validation scores: {:.2f}".format(scores_lr_cv.mean()))

        # lr on scaled data with CV
        pipe_lr = Pipeline([("scaler", StandardScaler()), ("linreg", LinearRegression())])
        scores_lr_sccv = cross_val_score(pipe_lr, bf_new, y)
        print("Cross-validation scores on scaled data: {}".format(scores_lr_sccv))
        print("Average cross-validation score on scaled: {:.2f}".format(scores_lr_sccv.mean()))
```

```
Results of Linear Regression
Training set score (R^2): 0.95
Test set score (R^2): 0.95
Training set score on scaled data (R^2): 0.95
Test set score on scaled data (R^2): 0.95
```

```
Cross-validation scores: [0.92  0.928 0.934]
Average Cross-validation scores: 0.93
Cross-validation scores on scaled data: [0.92  0.928 0.934]
Average cross-validation score on scaled: 0.93
```

Both non-scaled and scaled data give the same results with LinearRegression (95% training and test score, and 93% average cross-validation score), so we conclude that scaling data does not improve the performance of the model.

```
In [10]: from sklearn.linear_model import Ridge

         print("Results of Ridge")

         ridge = Ridge().fit(X_train, y_train)
         print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
         print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))

         ridge_sc = Ridge().fit(X_train_scaled, y_train)
         print("Training set score on scaled data: {:.2f}".format(ridge_sc.score(X_train_scale
         print("Test set score on scaled data: {:.2f}".format(ridge_sc.score(X_test_scaled, y_

         ridge_cv = Ridge()
         scores_r_cv = cross_val_score(ridge_cv, bf_new, y)
         print("Cross-validation scores: {}".format(scores_r_cv))
         print("Average cross-validation score: {:.2f}".format(scores_r_cv.mean()))

         # ridge on scaled data with CV
         pipe_r = Pipeline([("scaler", StandardScaler()), ("ridge", Ridge())])
         scores_r_sccv = cross_val_score(pipe_r, bf_new, y)
         print("Cross-validation scores: {}".format(scores_r_sccv))
         print("Average cross-validation score on scaled: {:.2f}".format(scores_r_sccv.mean())
```

```
Results of Ridge
Training set score: 0.88
Test set score: 0.90
Training set score on scaled data: 0.95
Test set score on scaled data: 0.95
Cross-validation scores: [0.833 0.877 0.857]
Average cross-validation score: 0.86
Cross-validation scores: [0.92  0.928 0.934]
Average cross-validation score on scaled: 0.93
```

Unlike LinearRegression, there seems to be a lot of improvement in Ridge when the data is scaled. After scaling, training and test scores are increased 7% and 5%, respectively. Furthermore, average cross-validation score also sees 9% increase.

```
In [11]: from sklearn.linear_model import Lasso
```

```
lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))


lasso_sc = Lasso().fit(X_train_scaled, y_train)
print("Training set score on scaled data: {:.2f}".format(lasso_sc.score(X_train_scale
print("Test set score on scaled data: {:.2f}".format(lasso_sc.score(X_test_scaled, y_


lasso_cv = Lasso()
scores_lasso_cv = cross_val_score(lasso_cv, bf_new, y)
print("Cross-validation scores: {}".format(scores_lasso_cv))
print("Average cross-validation score: {:.2f}".format(scores_lasso_cv.mean()))

# ridge on scaled data with CV
pipe_l = Pipeline([("scaler", StandardScaler()), ("lasso", Lasso())])
scores_l_sccv = cross_val_score(pipe_l, bf_new, y)
print("Cross-validation scores: {}".format(scores_l_sccv))
print("Average cross-validation score on scaled: {:.2f}".format(scores_l_sccv.mean())
```

```
Training set score: 0.85
Test set score: 0.87
Number of features used: 5
Training set score on scaled data: 0.95
Test set score on scaled data: 0.95
Cross-validation scores: [0.804 0.848 0.828]
Average cross-validation score: 0.83
Cross-validation scores: [0.917 0.925 0.939]
Average cross-validation score on scaled: 0.93
```

Similarly to Ridge, Lasso also performs much better with scaled data. After scaling, training and test set scores are increased for 10% and 8%, while the average cross-validation score is increased for 10%.

```
In [10]: from sklearn.neighbors import KNeighborsRegressor

reg = KNeighborsRegressor()
reg.fit(X_train, y_train)
print("Training set accuracy: {:.2f}".format(reg.score(X_train, y_train)))
print("Test set accuracy: %.2f" % reg.score(X_test, y_test))

reg_sc = KNeighborsRegressor()
reg_sc.fit(X_train_scaled, y_train)
print("Training set accuracy on scaled data: {:.2f}".format(reg_sc.score(X_train_scal
print("Test set accuracy on scaled data: %.2f" % reg_sc.score(X_test_scaled, y_test))
```

```
        scores_clf_cv = cross_val_score(reg_sc, bf_new, y)
        print("Test set accuracy on cross validation: {}".format(scores_clf_cv))
        print("Average cross-validation score: {:.2f}".format(scores_clf_cv.mean()))

        # ridge on scaled data with CV
        pipe_knn = Pipeline([("scaler", StandardScaler()), ("knn", KNeighborsRegressor())])
        scores_knn_sccv = cross_val_score(pipe_knn, bf_new, y)
        print("Cross-validation scores: {}".format(scores_knn_sccv))
        print("Average cross-validation score on scaled: {:.2f}".format(scores_knn_sccv.mean()
```

```
Out[10]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                  metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                  weights='uniform')

Training set accuracy: 0.80
Test set accuracy: 0.74
```

```
Out[10]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                  metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                  weights='uniform')

Training set accuracy on scaled data: 0.92
Test set accuracy on scaled data: 0.90
Test set accuracy on cross validation: [0.468 0.665 0.46 ]
Average cross-validation score: 0.53
Cross-validation scores: [0.842 0.867 0.757]
Average cross-validation score on scaled: 0.82
```

From the results above, it seems that kNN's performance also improves with scaled data. Training and test set accuracy improve for 12% and 16% after scaling, respectively. Furthermore, average cross-validation score is also improved by 29%.

```
In [15]: from sklearn.linear_model import ElasticNet

        en = ElasticNet().fit(X_train, y_train)
        print("Training set score: {:.2f}".format(en.score(X_train, y_train)))
        print("Test set score: {:.2f}".format(en.score(X_test, y_test)))
        # print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))


        en_sc = ElasticNet().fit(X_train_scaled, y_train)
        print("Training set score on scaled data: {:.2f}".format(en_sc.score(X_train_scaled,
        print("Test set score on scaled data: {:.2f}".format(en_sc.score(X_test_scaled, y_tes


        en_cv = ElasticNet()
```

```
        scores_en_cv = cross_val_score(en_cv, bf_new, y)
        print("Cross-validation scores: {}".format(scores_en_cv))
        print("Average cross-validation score: {:.2f}".format(scores_lasso_cv.mean()))

        # ridge on scaled data with CV
        pipe_en = Pipeline([("scaler", StandardScaler()), ("en", ElasticNet())])
        scores_en_sccv = cross_val_score(pipe_en, bf_new, y)
        print("Cross-validation scores: {}".format(scores_en_sccv))
        print("Average cross-validation score on scaled: {:.2f}".format(scores_en_sccv.mean()))
```

```
Training set score: 0.85
Test set score: 0.87
Training set score on scaled data: 0.91
Test set score on scaled data: 0.91
Cross-validation scores: [0.803 0.849 0.828]
Average cross-validation score: 0.83
Cross-validation scores: [0.879 0.885 0.894]
Average cross-validation score on scaled: 0.89
```

ElasticNet is another method that also performs better with scaled data. Training and test set score are improved by 6% and 4% increase, respectively, after scaling data, and the average cross-validation score is improved by 6%.

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.

In [13]:
```
# k-fold or shuffle-split fucntion is meant here
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

from sklearn.model_selection import ShuffleSplit

# KFold(n_splits=3, shuffle=False, random_state=None)
sp = ShuffleSplit(n_splits=10, random_state=0)


figs, axs = plt.subplots(3,1)
ridge0001 = Ridge(alpha=0.001).fit(X_train, y_train)
ridge0005 = Ridge(alpha=0.005).fit(X_train, y_train)
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
ridge001 = Ridge(alpha=0.01).fit(X_train, y_train)
ridge1 = Ridge(alpha=1).fit(X_train, y_train)
```

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
ridge100 = Ridge(alpha=100).fit(X_train, y_train)

# ridge_train = np.array([[ridge0001.score(X_train, y_train), ridge0005.score(X_train
#                          ridge001.score(X_train, y_train), ridge01.score(X_train, y_
#                          ridge1.score(X_train, y_train, ridge10.score(X_train, y_tra

ridge_train = np.array([[ridge0001.score(X_train, y_train), ridge0005.score(X_train, y
                        ridge001.score(X_train, y_train), ridge01.score(X_train, y_tra
                        ridge1.score(X_train, y_train), ridge10.score(X_train, y_trai
                        ridge100.score(X_train, y_train)]])

ridge_test = np.array([[ridge0001.score(X_test, y_test), ridge0005.score(X_test, y_tes
                       ridge001.score(X_test, y_test), ridge01.score(X_test, y_test),
                        ridge10.score(X_test, y_test), ridge100.score(X_test, y_test)]

test = pd.Series([ridge0001.score(X_test, y_test), ridge0005.score(X_test, y_test),
                        ridge001.score(X_train, y_train), ridge01.score(X_train, y_tra
                        ridge1.score(X_train, y_train), ridge10.score(X_train, y_trai
                         index=[0.001, 0.005, 0.01, 0.1, 1, 10, 100])

# sns.tsplot(ridge_train.set_index('alpha')['score'], ax=axs[0], err_style=None)
ax2 = sns.tsplot(data=ridge_train, time=[0.001, 0.005, 0.01, 0.1, 1, 10, 100],  ax=axs
sns.tsplot(ridge_test,time=[0.001, 0.005, 0.01, 0.1, 1, 10, 100], color = 'r', ax=axs
ax2.set(xlabel='Alpha value', ylabel='R^2 score')


ridge0001 = Ridge(alpha=0.001).fit(X_train, y_train)
ridge0005 = Ridge(alpha=0.005).fit(X_train, y_train)
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
ridge001 = Ridge(alpha=0.01).fit(X_train, y_train)
ridge1 = Ridge(alpha=1).fit(X_train, y_train)


ridge_train = np.array([[ridge0001.score(X_train, y_train), ridge0005.score(X_train, y
                        ridge001.score(X_train, y_train), ridge01.score(X_train, y_tra
                        ridge1.score(X_train, y_train)]])

ridge_test = np.array([[ridge0001.score(X_test, y_test), ridge0005.score(X_test, y_tes
                       ridge001.score(X_test, y_test), ridge01.score(X_test, y_test),

test = pd.Series([ridge0001.score(X_test, y_test), ridge0005.score(X_test, y_test),
                        ridge001.score(X_train, y_train), ridge01.score(X_train, y_tra
                        ridge1.score(X_train, y_train)],
                         index=[0.001, 0.005, 0.01, 0.1, 1])

# sns.tsplot(ridge_train.set_index('alpha')['score'], ax=axs[0], err_style=None)
ax3 = sns.tsplot(data=ridge_train, time=[0.001, 0.005, 0.01, 0.1, 1],  ax=axs[1], err
```

```
sns.tsplot(ridge_test,time=[0.001, 0.005, 0.01, 0.1, 1], color = 'r', ax=axs[1], err_s
ax3.set(xlabel='Alpha value', ylabel='R^2 score')
```

```
lasso0001 = Lasso(alpha=0.001, max_iter=100000).fit(X_train, y_train)
lasso0005 = Lasso(alpha=0.005, max_iter=100000).fit(X_train, y_train)
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
lasso01 = Lasso(alpha=0.1, max_iter=100000).fit(X_train, y_train)
lasso1 = Lasso(alpha=1, max_iter=100000).fit(X_train, y_train)
lasso10 = Lasso(alpha=10, max_iter=100000).fit(X_train, y_train)
lasso100 = Lasso(alpha=100, max_iter=100000).fit(X_train, y_train)

lasso_train = np.array([[lasso0001.score(X_train, y_train), lasso0005.score(X_train, y
                         lasso001.score(X_train, y_train), lasso01.score(X_train, y_tra
                         lasso1.score(X_train, y_train),
                         lasso10.score(X_train, y_train), lasso100.score(X_train, y_tra
lasso_test = np.array([[lasso0001.score(X_test, y_test), lasso0005.score(X_test, y_tes
                        lasso001.score(X_test, y_test), lasso01.score(X_test, y_test),
                         lasso10.score(X_test, y_test), lasso100.score(X_test, y_test)]

ax1 = sns.tsplot(data=lasso_train, time=[0.001, 0.005, 0.01, 0.1, 1, 10, 100],  ax=axs
sns.tsplot(lasso_test,time=[0.001, 0.005, 0.01, 0.1, 1, 10, 100], color = 'r', ax=axs
ax1.set(xlabel='Alpha value', ylabel='R^2 score')
```

```
Out[13]: Text(0.5,1,'Ridge performance')
```

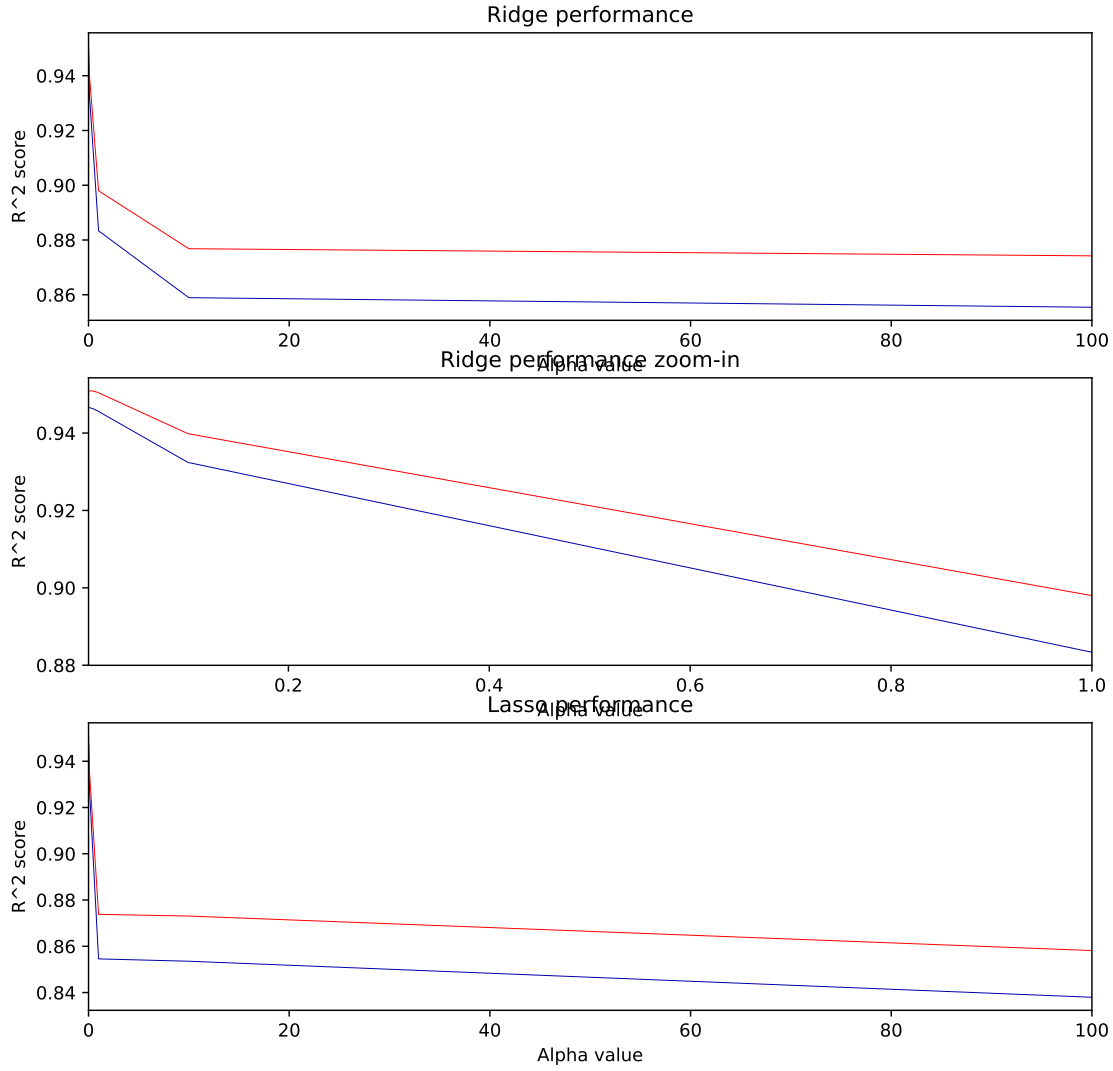
```
Out[13]: [Text(0,0.5,'R^2 score'), Text(0.5,0,'Alpha value')]
```

```
Out[13]: Text(0.5,1,'Ridge performance zoom-in')
```

```
Out[13]: [Text(0,0.5,'R^2 score'), Text(0.5,0,'Alpha value')]
```

```
Out[13]: Text(0.5,1,'Lasso performance')
```

```
Out[13]: [Text(0,0.5,'R^2 score'), Text(0.5,0,'Alpha value')]
```



Two graphs are plotted for ridge perfomance as the second graph shows a zoom-in of alpha value between 0 and 1.0.

From this graph we can see clearly that low alpha value - less regularization results in higher R^2 score of the test set. However, alpha = 0 and any low value of alpha means that the regularization is almost not present, and as such it is quite likely that the model overfits the data. After alpha=10 we observe that the R^2 score 'stabilizes', which makes us think that the model for alpha>= 10 is relatively general and might be rather underfitting. So accroding to us the optimal performance of Ridge is a model with alpha value between 1 and 10.

When creating a Lasso model with alpha value smaller than 1, we observe similarly to Ridge that the smaller the regularization is the higher R^2 is observed. But this effect is likely to be a result from overfitting. That is why we consider that an optimal performing Lasso model has alpha value, between 10 and 40. After alpha = 40, the model starts getting more conservative.

```
In [10]: from sklearn.model_selection import GridSearchCV
         from sklearn.linear_model import ElasticNet
```

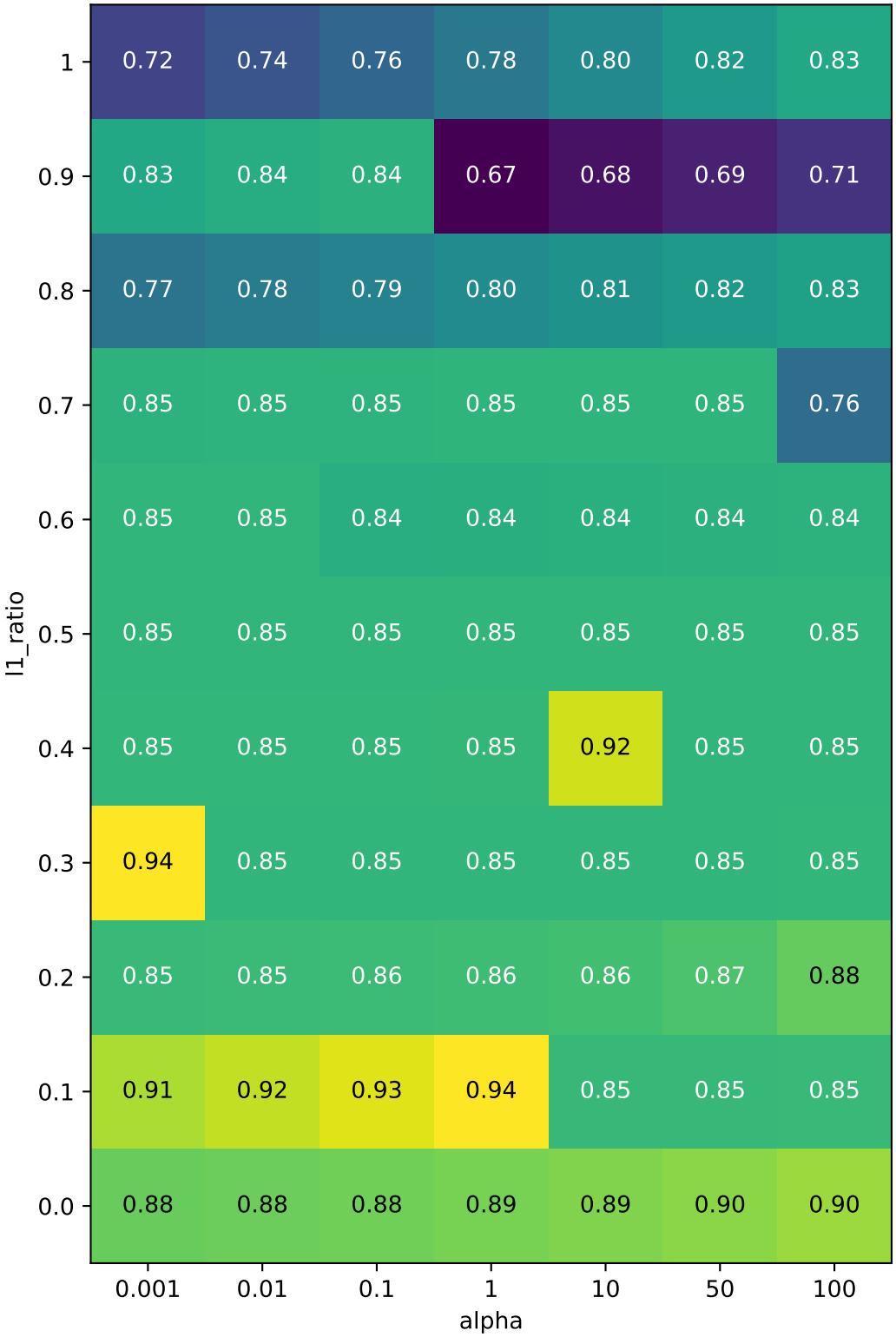
```
param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 50, 100],
              'l1_ratio': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]}
print("Parameter grid:\n{}".format(param_grid))


grid_search = GridSearchCV(ElasticNet(), param_grid, cv=10)
grid_search.fit(X_train, y_train)

results = pd.DataFrame(grid_search.cv_results_)
# Show the first 5 rows
display(results.head())
grid_search = GridSearchCV(ElasticNet(), param_grid, cv=10)
grid_search.fit(X_train, y_train)

results = pd.DataFrame(grid_search.cv_results_)

scores = np.array(results.mean_test_score).reshape(11, 7)
# plot the mean cross-validation scores
mglearn.tools.heatmap(scores, xlabel='alpha', xticklabels=param_grid['alpha'],
                      ylabel='l1_ratio', yticklabels=param_grid['l1_ratio'], cmap="vi
```

```
Parameter grid:
{'alpha': [0.001, 0.01, 0.1, 1, 10, 50, 100], 'l1_ratio': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, (
```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	\
0	9.38e-03	0.0	0.88	0.89	
1	1.56e-03	0.0	0.88	0.89	
2	1.56e-03	0.0	0.88	0.89	
3	1.56e-03	0.0	0.89	0.89	
4	3.13e-03	0.0	0.89	0.90	

	...	std_fit_time	std_score_time	std_test_score	std_train_score
0	...	7.66e-03	0.0	0.02	2.31e-03
1	...	4.69e-03	0.0	0.02	2.27e-03
2	...	4.69e-03	0.0	0.02	2.23e-03
3	...	4.69e-03	0.0	0.02	2.17e-03
4	...	6.25e-03	0.0	0.02	2.11e-03

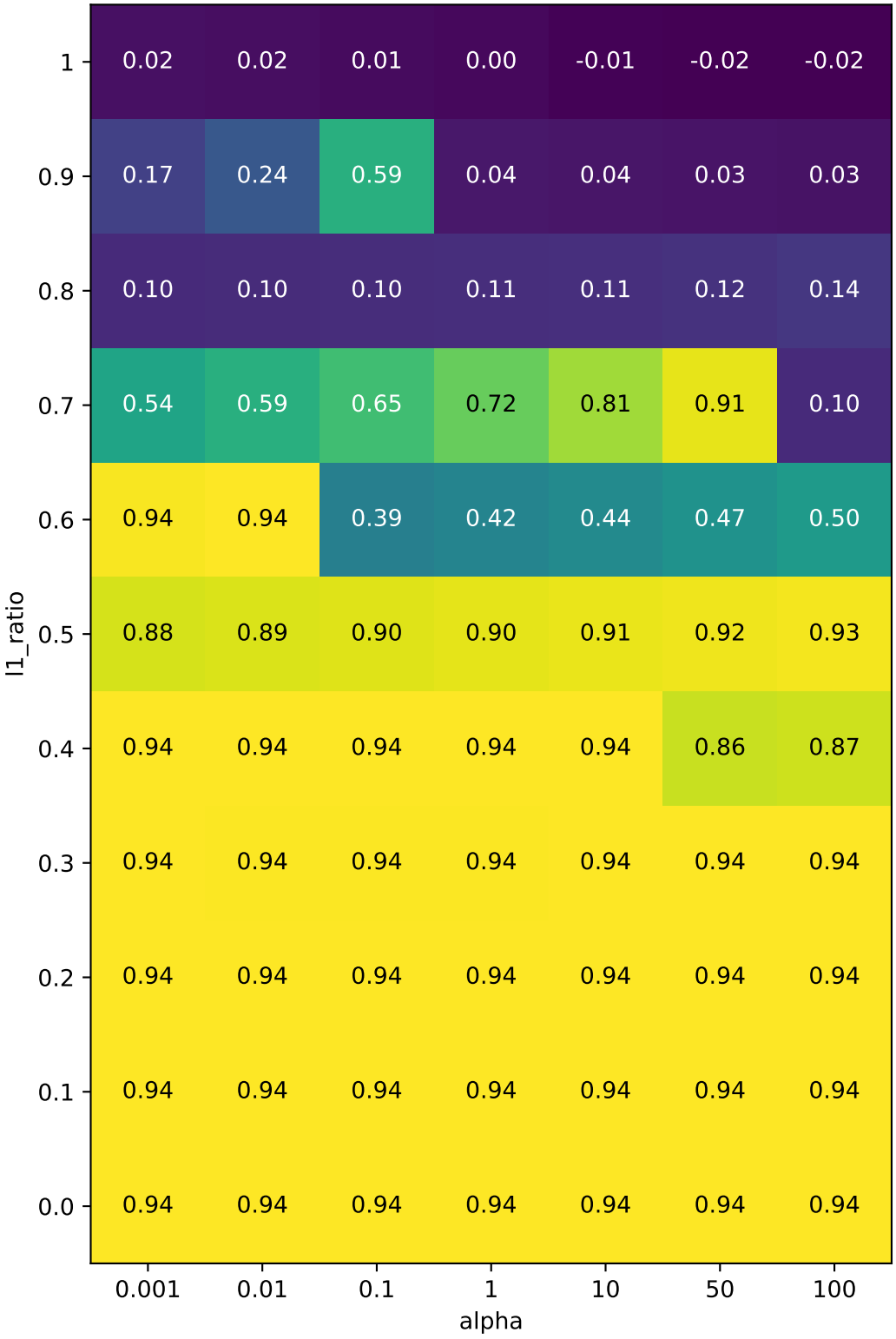
[5 rows x 32 columns]




```
In [11]: grid_search.fit(X_train, y_train)
```

```
        results = pd.DataFrame(grid_search.cv_results_)
        # Show the first 5 rows
        display(results.head())
        grid_search = GridSearchCV(ElasticNet(), param_grid, cv=10)
        grid_search.fit(X_train_scaled, y_train)

        results = pd.DataFrame(grid_search.cv_results_)

        scores = np.array(results.mean_test_score).reshape(11, 7)
        # plot the mean cross-validation scores
        mglearn.tools.heatmap(scores, xlabel='alpha', xticklabels=param_grid['alpha'],
                              ylabel='l1_ratio', yticklabels=param_grid['l1_ratio'], cmap="vi
```

```
   mean_fit_time  mean_score_time  mean_test_score  mean_train_score  \
0       6.25e-03              0.0             0.88              0.89
1       3.13e-03              0.0             0.88              0.89
2       0.00e+00              0.0             0.88              0.89
3       1.56e-03              0.0             0.89              0.89
4       0.00e+00              0.0             0.89              0.90

        ...        std_fit_time std_score_time std_test_score  std_train_score
0       ...            7.66e-03           0.0           0.02         2.31e-03
1       ...            6.25e-03           0.0           0.02         2.27e-03
2       ...            0.00e+00           0.0           0.02         2.23e-03
3       ...            4.69e-03           0.0           0.02         2.17e-03
4       ...            0.00e+00           0.0           0.02         2.11e-03

[5 rows x 32 columns]
```



From the heat map we notice that the optimal performance of the elastic net without scaling is

best at l1_ratio=0.1 and alpha=0.94. However, there is a big difference with the heap mat generated from the scaled data, where the accuracy scores are boosted. The optimal performance of the Elastic Net with scaled data is for l1_ratio<= 0.4 and alpha<=10.

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. Does it agree with the linear models? What would look for when scouting for a baseball player?

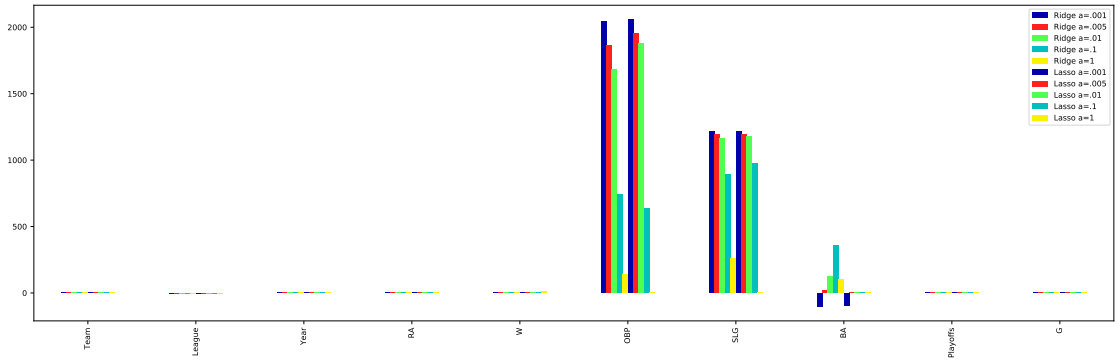
```
In [38]: from sklearn.ensemble import RandomForestClassifier

         forest = RandomForestClassifier(n_jobs=-1, random_state=0)
         forest.fit(X_train, y_train)
         print("Features importances: {}".format(forest.feature_importances_))
```

```
Out[38]: 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=0, verbose=0, warm_start=False)

Features importances: [0.115 0.036 0.128 0.136 0.13  0.125 0.145 0.13  0.015 0.04 ]
```

```
In [57]: scores = pd.DataFrame(columns=bf_new.columns)
         scores.loc["Ridge a=.001"] = ridge0001.coef_
         scores.loc["Ridge a=.005"] = ridge0005.coef_
         scores.loc["Ridge a=.01"] = ridge001.coef_
         scores.loc["Ridge a=.1"] = ridge01.coef_
         scores.loc["Ridge a=1"] = ridge1.coef_
         scores.loc["Lasso a=.001"] = lasso0001.coef_
         scores.loc["Lasso a=.005"] = lasso0005.coef_
         scores.loc["Lasso a=.01"] = lasso001.coef_
         scores.loc["Lasso a=.1"] = lasso01.coef_
         scores.loc["Lasso a=1"] = lasso1.coef_
         display(scores)

         scores = scores.T
         ax = scores.plot(figsize=(20,6), kind='bar')
```

	Team	League	Year	RA	...	SLG	BA	Playoffs	\
Ridge a=.001	3.38e-02	-4.66	-0.35	0.27	...	1215.80	-102.10	3.12	
Ridge a=.005	3.23e-02	-4.63	-0.34	0.28	...	1191.60	21.39	3.31	
Ridge a=.01	3.01e-02	-4.62	-0.32	0.30	...	1166.19	126.69	3.48	
Ridge a=.1	-9.40e-03	-5.02	-0.17	0.48	...	894.32	355.95	4.08	
Ridge a=1	-1.09e-01	-6.29	0.16	0.76	...	261.99	105.25	3.92	
Lasso a=.001	3.38e-02	-4.65	-0.35	0.26	...	1213.94	-97.20	3.11	
Lasso a=.005	3.23e-02	-4.61	-0.34	0.27	...	1190.52	-0.00	3.19	
Lasso a=.01	3.00e-02	-4.62	-0.33	0.29	...	1179.11	0.00	3.17	
Lasso a=.1	-8.59e-03	-4.70	-0.18	0.51	...	974.12	0.00	2.81	

```
Lasso a=1    -1.37e-01   -2.54  0.28  0.87  ...      0.00    0.00      0.00

                 G
Ridge a=.001  4.54
Ridge a=.005  4.39
Ridge a=.01   4.22
Ridge a=.1    2.82
Ridge a=1     0.93
Lasso a=.001  4.56
Lasso a=.005  4.47
Lasso a=.01   4.35
Lasso a=.1    2.12
Lasso a=1    -0.00

[10 rows x 10 columns]
```



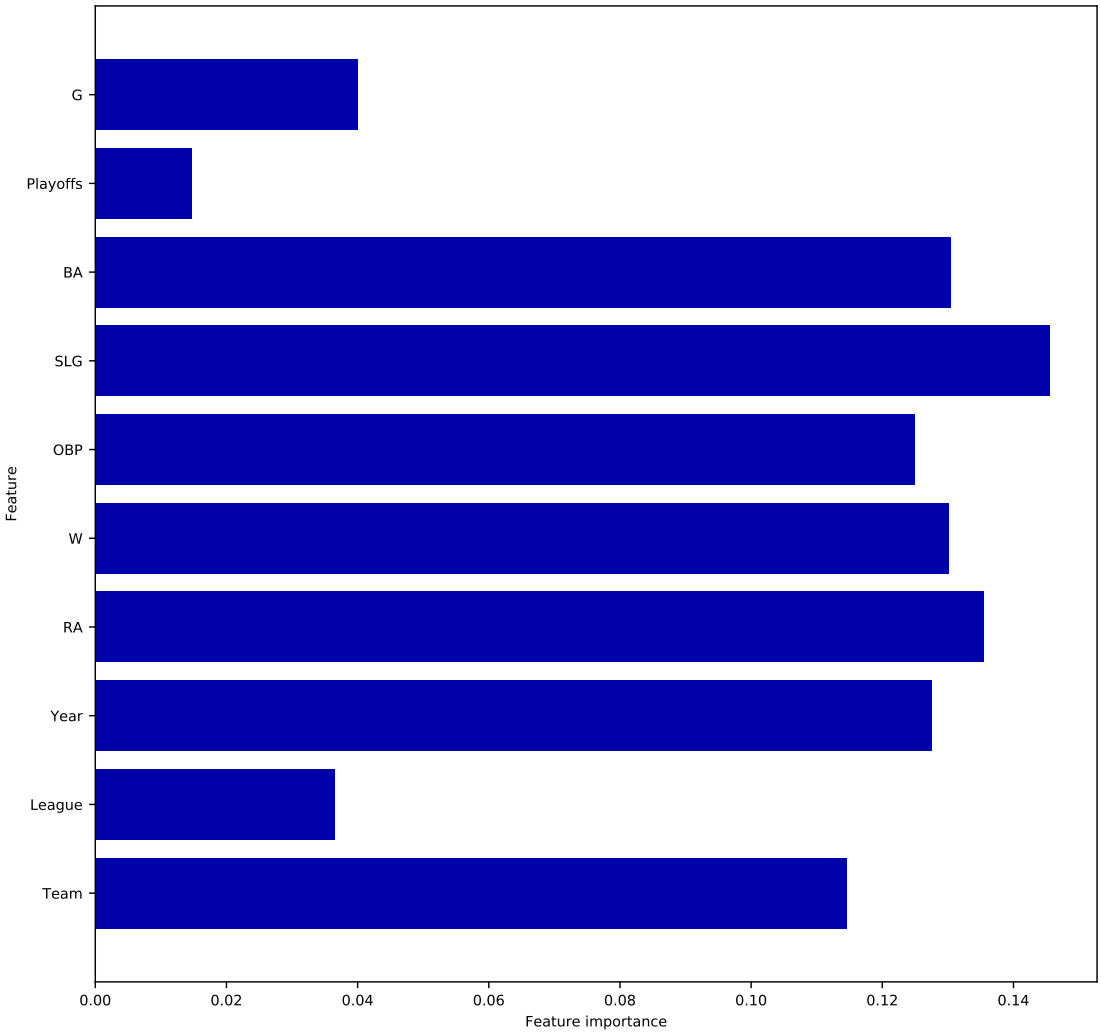

```
In [58]: n_features = bf_new.shape[1]
         plt.barh(range(n_features), forest.feature_importances_, align='center')
         plt.yticks(np.arange(n_features), bf_new.columns)
         plt.xlabel("Feature importance")
         plt.ylabel("Feature")
         plt.ylim(-1, n_features)

Out[58]: <Container object of 10 artists>

Out[58]: ([<matplotlib.axis.YTick at 0x2e728e87400>,
           <matplotlib.axis.YTick at 0x2e728f5c438>,
           <matplotlib.axis.YTick at 0x2e728d3ce80>,
           <matplotlib.axis.YTick at 0x2e728d14240>,
           <matplotlib.axis.YTick at 0x2e728d147f0>,
           <matplotlib.axis.YTick at 0x2e727b727f0>,
           <matplotlib.axis.YTick at 0x2e727b72710>,
           <matplotlib.axis.YTick at 0x2e727b72860>,
```

```
 <matplotlib.axis.YTick at 0x2e727985cf8>,
 <matplotlib.axis.YTick at 0x2e727985240>],
<a list of 10 Text yticklabel objects>)
```

```
Out[58]: Text(0.5,0,'Feature importance')
```

```
Out[58]: Text(0,0.5,'Feature')
```

```
Out[58]: (-1, 10)
```



The optimized models seem to mostly agree that either SLG or OBP is the most important feature. Even though there is no clear agreement, we see that models that see OBP as most important also see SLG as the second most important. RandomForest itself sees SLG as the most important, which is aligned with some optimized linear models (Ridge .1, Lasso .1, and Lasso 1).

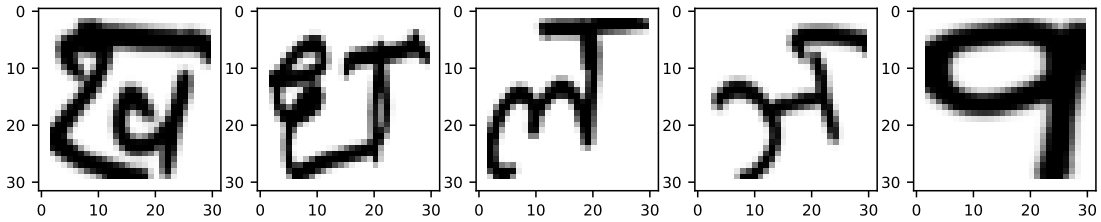
After examining these results, we conclude that it is important to look for SLG and OBP when scouting for baseball players.

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 [5]: 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
```

```
In [4]: 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();
```



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 [6]: from sklearn.model_selection import StratifiedShuffleSplit
        from sklearn.linear_model import LogisticRegression
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.ensemble import RandomForestClassifier

        # X_train, X_test, y_train, y_test = train_test_split(bf_new, y, random_state=0)
        # test + train data should be a subsample of 10%, preserving 75% for training and 25%
        # using stratified shuffle split will result in train size = 0.075 of the initial data
        strat_shuffle_cv = StratifiedShuffleSplit(n_splits=1, test_size=0.025, train_size=0.075
```

```
        results_lr = ([])
        results_knn = ([])
        results_rf = ([])

        for train_index, test_index in strat_shuffle_cv.split(X, y):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            log_reg = LogisticRegression().fit(X_train, y_train)
            print("Training set score logistic regression: {:.2f}".format(log_reg.score(X_trai
            result_lr = log_reg.score(X_test, y_test)
            print("Test set score Logistic regression: {:.2f}".format(result_lr))
            results_lr = np.append(result_lr, results_lr)

            knn = KNeighborsClassifier()
            knn.fit(X_train, y_train)
            result_knn = knn.predict(X_test)==y_test
            print("Train set predictions kNN:\n {}".format(np.mean(knn.predict(X_train)==y_tra
            print("Test set predictions kNN:\n {}".format(np.mean(result_knn)))
            results_knn = np.append(result_knn, results_knn)


            forest = RandomForestClassifier(n_jobs=-1, random_state=0)
            forest.fit(X_train, y_train)
            print("Accuracy on training set RF: {:.3f}".format(forest.score(X_train, y_train))
            print("Accuracy on test set RF: {:.3f}".format(forest.score(X_test, y_test)))
```

```
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.48
```

```
Out[6]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')
```

```
Train set predictions kNN:
 0.85
Test set predictions kNN:
 0.7573913043478261
```

```
Out[6]: 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=0, verbose=0, warm_start=False)
```

```
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.557
```

In principal, predictive accuracy is not a good measurement because high predictive accuracy can mean that the model is overfitting. In this case, we can see that LogisticRegression and RandomForest have predictive accuracy around 50%, which for classification problem is not good.

```
In [10]: from sklearn.model_selection import StratifiedShuffleSplit
         from sklearn.linear_model import LogisticRegression
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.ensemble import RandomForestClassifier
         import time

         def doLogReg(X_tr, X_t,  y_tr, y_t, lr_stop):
             total_running_time = 0
             start_time = time.time()
             log_reg = LogisticRegression(n_jobs=-1).fit(X_tr, y_tr)
             print("Training set score logistic regression: {:.2f}".format(log_reg.score(X_tr,
             result_lr = log_reg.score(X_t, y_t)
             print("Test set score Logistic regression: {:.2f}".format(result_lr))
             total_running_time = (time.time() - start_time)
             return round(result_lr, 2), round(total_running_time, 2)


         def doKnn(X_tr, X_t,  y_tr, y_t, knn_stop):
             total_running_time = 0
             start_time = time.time()
             knn = KNeighborsClassifier()
             knn.fit(X_tr, y_tr)
             result_knn = np.mean(knn.predict(X_test)==y_test)
             print("Train set predictions kNN: {}".format(np.mean(knn.predict(X_train)==y_trai
             print("Test set predictions kNN: {}".format(np.mean(result_knn)))
             #     results_knn = np.append(result_knn, results_knn)
             total_running_time = (time.time() - start_time)

             return round(result_knn, 2), round(total_running_time, 2)


         def doRF(X_tr, X_t,  y_tr, y_t, rf_stop):

             total_running_time = 0
             start_time = time.time()
             forest = RandomForestClassifier(n_jobs=-1, random_state=0)
             forest.fit(X_tr, y_tr)
             result_rf = forest.score(X_t, y_t)
             print("Accuracy on training set RF: {:.3f}".format(forest.score(X_tr, y_tr)))
             print("Accuracy on test set RF: {:.3f}".format(result_rf))
             total_running_time = (time.time() - start_time)
             return round(result_rf, 2), round(total_running_time, 2)
```

```
def loopSubsamples(subsample_size):

#     subsample_size = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25]

    train_sizes = ([])
    test_sizes = ([])

    results_lr = ([])
    running_time_lr = ([])
    lr_stop = False

    results_knn = ([])
    running_time_knn = ([])
    knn_stop = False

    results_rf = ([])
    running_time_rf = ([])
    rf_stop = False

    for percent_subs in subsample_size:
        strat_shuffle = StratifiedShuffleSplit(n_splits=1, test_size=(percent_subs*0.
                                               train_size=((percent_subs*0.75)/100),

        total_running_time = 0

        print("Results for subsample of \n {} %".format(percent_subs))

        for train_ind, test_ind in strat_shuffle.split(X, y):
            X_train, X_test = X[train_ind], X[test_ind]
            y_train, y_test = y[train_ind], y[test_ind]

            if not lr_stop:
                result_lr, total_time_lr = doLogReg(X_train, X_test, y_train, y_test,
                running_time_lr = np.append(running_time_lr, total_time_lr)
                results_lr = np.append(results_lr, result_lr)
                if total_time_lr > 600:
                    lr_stop = True

            if not knn_stop:
                result_knn, total_time_knn = doKnn(X_train, X_test, y_train, y_test,
                running_time_knn = np.append(running_time_knn, total_time_knn)
                results_knn = np.append(results_knn, result_knn)
                if total_time_knn > 600:
                    knn_stop = True

            if not rf_stop:
                result_rf, total_time_rf = doRF(X_train, X_test, y_train, y_test, rf_
                running_time_rf = np.append(total_time_rf, running_time_rf)
```

```
                    results_rf = np.append(results_rf, result_rf)
                    if total_time_rf > 600:
                        rf_stop = True

        return results_lr, running_time_lr, results_knn, running_time_knn, results_rf, ru

    subsample_size = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25]
    subsample_size_lr = [1,2 ,3 ,4, 5,6 ,7, 8, 9, 10, 15]

    results_lr, running_time_lr, results_knn, running_time_knn, results_rf, running_time_
    # type(results_lr)
    # data_lr = [results_lr, running_time_lr ]
    # type(data_lr)
    fig, axs = plt.subplots(3, 2, figsize=(10, 10))
    plt.tight_layout()
    # sns.tsplot(ridge_train.set_index('alpha')['score'], ax=axs[0], err_style=None)
    ax1 = sns.pointplot(x = subsample_size_lr, y = results_lr, ax=axs[0, 0], err_style=No
    ax1.set(xlabel='Subsample size in %', ylabel='R^2 score')

    ax2 = sns.pointplot(x = subsample_size_lr, y =running_time_lr, ax=axs[0, 1], err_style
    ax2.set(xlabel='Subsample size in %', ylabel='Running time')
    ax2.set_title("Results of Logistic regression")

    ax3 = sns.pointplot(x = subsample_size, y = results_knn, ax=axs[1, 0], err_style=None
    ax3.set(xlabel='Subsample size in %', ylabel='R^2 score')
    ax3.set_title("Results of kNN")

    ax4 = sns.pointplot(x = subsample_size, y =running_time_knn, ax=axs[1, 1], err_style=
    ax4.set(xlabel='Subsample size in %', ylabel='Running time')


    ax5 = sns.pointplot(x = subsample_size, y = results_rf, ax=axs[2, 0], err_style=None)
    ax5.set(xlabel='Subsample size in %', ylabel='R^2 score')
    ax5.set_title("Results of Random Forest")

    ax6 = sns.pointplot(x = subsample_size, y =running_time_rf,  ax=axs[2,1], err_style=N
    ax6.set(xlabel='Subsample size in %', ylabel='Running time')
    # sns.tsplot(ridge_test,time=[0.001, 0.005, 0.01, 0.1, 1], color = 'r', ax=axs[2], er
```

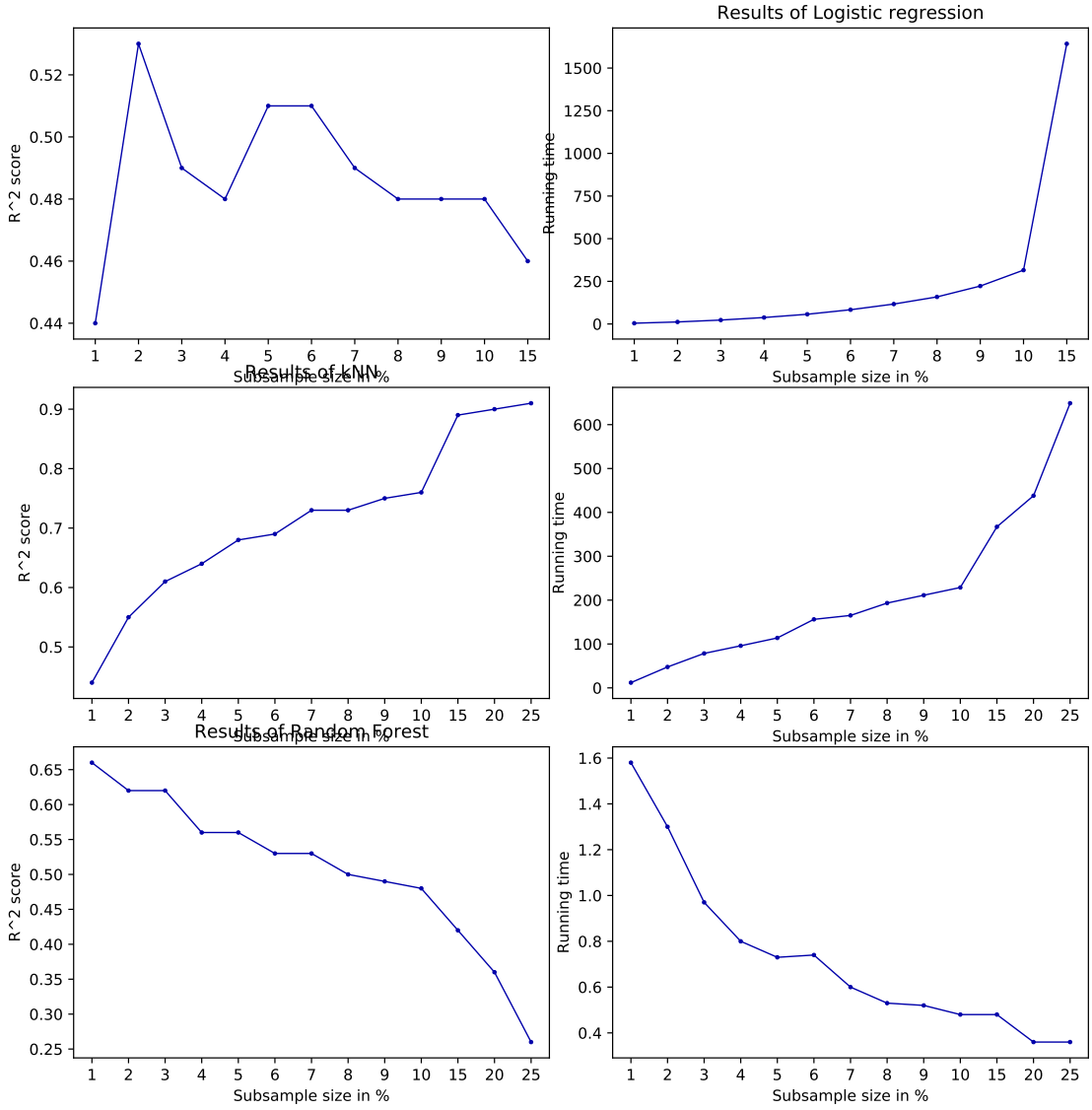
```
Results for subsample of
 1 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.44
Train set predictions kNN:
 0.45202898550724635
Test set predictions kNN:
 0.43869565217391304
Accuracy on training set RF: 0.993
```

```
Accuracy on test set RF: 0.257
Results for subsample of
 2 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.53
Train set predictions kNN:
 0.5797101449275363
Test set predictions kNN:
 0.5482608695652174
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.363
Results for subsample of
 3 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.49
Train set predictions kNN:
 0.637536231884058
Test set predictions kNN:
 0.6073913043478261
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.425
Results for subsample of
 4 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.48
Train set predictions kNN:
 0.6844927536231884
Test set predictions kNN:
 0.6426086956521739
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.477
Results for subsample of
 5 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.51
Train set predictions kNN:
 0.7284057971014493
Test set predictions kNN:
 0.6760869565217391
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.490
Results for subsample of
 6 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.51
Train set predictions kNN:
 0.76
Test set predictions kNN:
```

```
 0.6895652173913044
Accuracy on training set RF: 0.999
Accuracy on test set RF: 0.502
Results for subsample of
 7 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.49
Train set predictions kNN:
 0.715072463768116
Test set predictions kNN:
 0.7308695652173913
Accuracy on training set RF: 0.999
Accuracy on test set RF: 0.529
Results for subsample of
 8 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.47
Train set predictions kNN:
 0.8065217391304348
Test set predictions kNN:
 0.73
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.529
Results for subsample of
 9 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.48
Train set predictions kNN:
 0.8297101449275363
Test set predictions kNN:
 0.7478260869565218
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.556
Results for subsample of
 10 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.48
Train set predictions kNN:
 0.85
Test set predictions kNN:
 0.7573913043478261
Accuracy on training set RF: 0.999
Accuracy on test set RF: 0.557
Results for subsample of
 15 %
Training set score logistic regression: 1.00
Test set score Logistic regression: 0.46
Train set predictions kNN:
```

```
 0.8759420289855072
Test set predictions kNN:
 0.8860869565217391
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.618
Results for subsample of
 20 %
Train set predictions kNN:
 0.8917391304347826
Test set predictions kNN:
 0.9017391304347826
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.623
Results for subsample of
 25 %
Train set predictions kNN:
 0.9060869565217391
Test set predictions kNN:
 0.9095652173913044
Accuracy on training set RF: 0.998
Accuracy on test set RF: 0.660
```

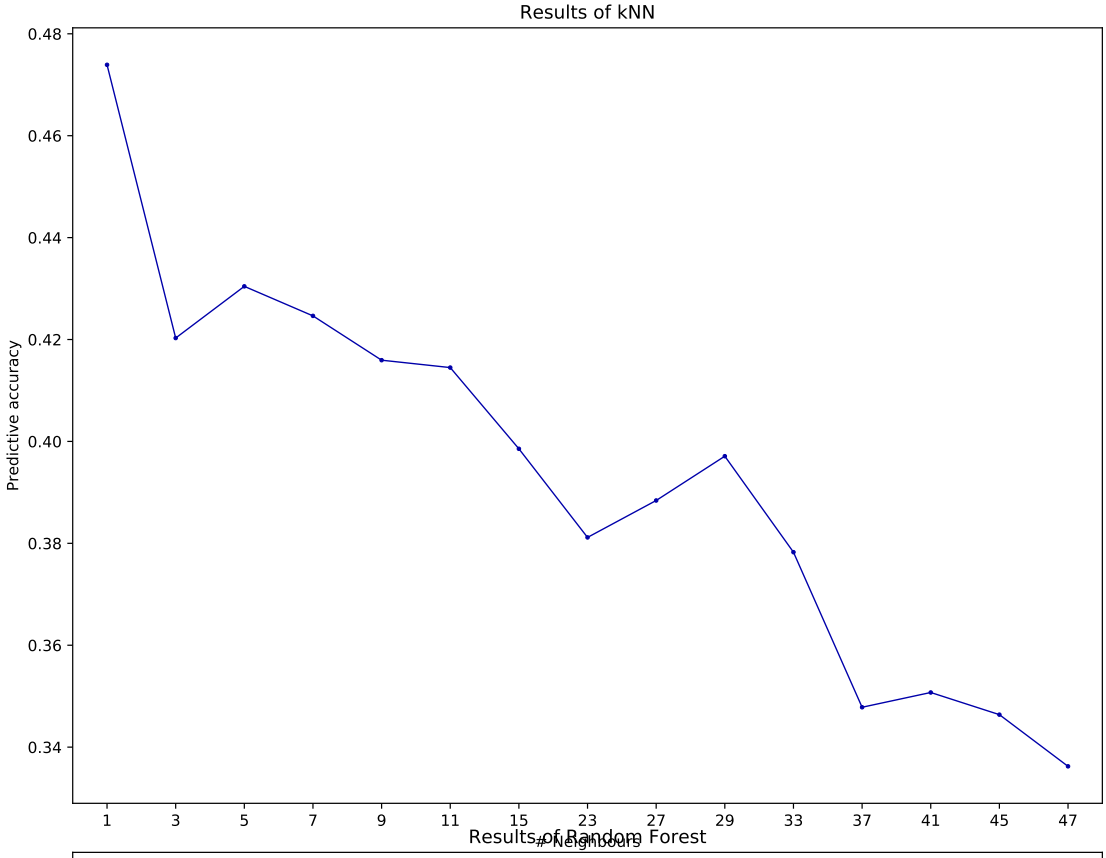
```
Out[10]: [Text(63.125,0.5,'R^2 score'), Text(0.5,452.718,'Subsample size in %')]

Out[10]: [Text(367.489,0.5,'Running time'), Text(0.5,452.718,'Subsample size in %')]

Out[10]: Text(0.5,1,'Results of Logistic regression')

Out[10]: [Text(63.125,0.5,'R^2 score'), Text(0.5,260.859,'Subsample size in %')]

Out[10]: Text(0.5,1,'Results of kNN')

Out[10]: [Text(367.489,0.5,'Running time'), Text(0.5,260.859,'Subsample size in %')]

Out[10]: [Text(63.125,0.5,'R^2 score'), Text(0.5,69,'Subsample size in %')]

Out[10]: Text(0.5,1,'Results of Random Forest')

Out[10]: [Text(367.489,0.5,'Running time'), Text(0.5,69,'Subsample size in %')]
```

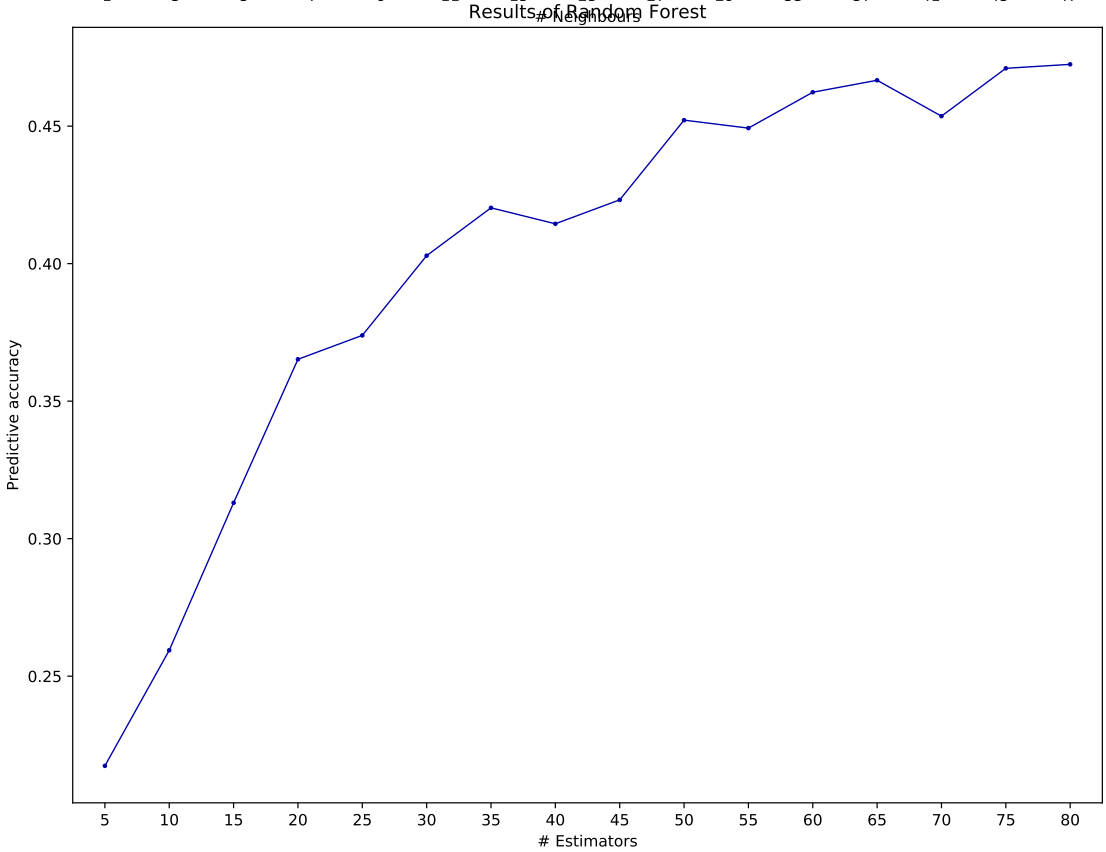


2 . Optimize the value for the number of neighbors k (keep $k < 50$) and the number of trees (keep *n_estimators* < 100) on the stratified 10% subsample. - Use 10-fold crossvalidation and plot k and *n_estimators* against the predictive accuracy. Which value of k, *n_estimators* should you pick?

```
In [34]: from sklearn.model_selection import StratifiedKFold
         from sklearn.model_selection import GridSearchCV
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.model_selection import StratifiedShuffleSplit
         from sklearn.model_selection import train_test_split
         from sklearn.model_selection import cross_val_score

         def grid_Search():
             accuracy_knn = ([])
```

```
    accuracy_rf= ([])

    neighbours = [1, 3,  5, 7, 9, 11, 15, 29, 23, 27, 33, 37, 41, 45, 47, 49]
    estimators = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85]

    param_grid = {'n_neighbors': neighbours}
    grid_search1 = GridSearchCV(KNeighborsClassifier(n_jobs=-1), param_grid, cv=10, n

    param_grid1 = {'n_estimators': estimators}
    grid_search2 = GridSearchCV(RandomForestClassifier(n_jobs=-1), param_grid1, cv=10


    X_u, X_n, y_u, y_n = train_test_split(X, y, train_size=0.01, random_state=0, strat
    X_train, X_test, y_train, y_test = train_test_split(X_u, y_u, random_state=0, stra


    grid_search1.fit(X_train, y_train)
    grid_search1.score(X_test, y_test)
    results = pd.DataFrame(grid_search1.cv_results_)
    scores = np.array(results.mean_test_score)
    scores_mean = np.array(scores).reshape(len(neighbours))

    fig, axs = plt.subplots(2, 1, figsize=(10, 15))
    plt.tight_layout()

    ax1 = sns.pointplot(x = param_grid['n_neighbors'], y = scores_mean, xticklabels=pa
                        ax=axs[0], err_style=None)
    ax1.set(xlabel='# Neighbours', ylabel='Predictive accuracy' )
    ax1.set_title("Results of kNN")

    grid_search2.fit(X_train, y_train)
    grid_search2.score(X_test, y_test)
    results2 = pd.DataFrame(grid_search2.cv_results_)
    scores2 = np.array(results2.mean_test_score)
    scores_mean2 = np.array(scores2).reshape(len(estimators))

    ax2 = sns.pointplot(x = param_grid1['n_estimators'], y = scores_mean2, xticklabels
                        ax=axs[1], err_style=None)
    ax2.set(xlabel='# Estimators', ylabel='Predictive accuracy' )
    ax2.set_title("Results of Random Forest")

grid_Search()
```


Results of kNN
Predictive accuracy
0.48
0.46
0.44
0.42
0.40
0.38
0.36
0.34
1
3
5
7
9
11
15
23
27
29
33
37
41
45
47
Neighbours


Results of Random Forest
Predictive accuracy
0.45
0.40
0.35
0.30
0.25
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
Estimators

From the graphs above, predictive accuracy is decreasing with more number of neighbors for kNN but increasing with more number of estimators for RandomForest. In general, we do not want to have too high accuracy because there is increased chance of overfitting. For kNN, we see that after 11 nearest neighbors the accuracy decreased sharply, so we think that 11 nearest neighbors would give the best accuracy without overfitting. As for RandomForest, we see that after 20 estimators the increase of accuracy is not so sharp anymore, so we think that 20 estimators will give the most optimal results.

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?

```
In [6]: from sklearn.model_selection import GridSearchCV
        from sklearn.model_selection import cross_val_score
        from sklearn.model_selection import RandomizedSearchCV
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.model_selection import train_test_split

        param_grid = {'max_features': np.arange(0.1,1.1,0.1),
                      'n_estimators': np.arange(10,110,10)}
        print(param_grid)
        # Get training and test data from 5% of the actual data
        X_train_10, X_test_10, y_train_10, y_test_10 = train_test_split(X, y, random_state=0, t
        X_train, X_test, y_train, y_test = train_test_split(X_train_10, y_train_10, random_stat
```

```
{'n_estimators': array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]), 'max_features': ar
```

```
In [5]: random_scores = cross_val_score(RandomizedSearchCV(RandomForestClassifier(), param_dist
                                                n_iter=2, n_jobs =-1), X_train, y_train, cv=5, n_jo
        print("Cross-validation scores: {}".format(random_scores))
```

```
Cross-validation scores: [0.585 0.623 0.667 0.644 0.643]
Average cross-validation score on scaled: 0.63
```

```
In [ ]: random_scores = cross_val_score(GridSearchCV(RandomForestClassifier(), param_grid, cv=
                                        X_train, y_train, cv=5, n_jobs=-1)
        print("Cross-validation scores: {}".format(random_scores))
```

TODO: enter comment here

```
In [ ]: grid_cv = GridSearchCV(RandomForestClassifier(), param_grid, cv=3, n_jobs=-1)
        grid_cv.fit(X_train, y_train)
```

```
---------------------------------------------------------------------------

KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-6-ebd1428085d5> in <module>()
  1 grid_cv = GridSearchCV(RandomForestClassifier(), param_grid, cv=3, n_jobs=-1)
----> 2 grid_cv.fit(X_train, y_train)


c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\sklearn\model_selection\_sea
637                                     error_score=self.error_score)
638           for parameters, (train, test) in product(candidate_params,
--> 639                                                    cv.split(X, y, groups)))
640
641         # if one choose to see train score, "out" will contain train score info


c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\sklearn\externals\joblib\par
787                 # consumption.
788                 self._iterating = False
--> 789             self.retrieve()
790             # Make sure that we get a last message telling us we are done
791             elapsed_time = time.time() - self._start_time


c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\sklearn\externals\joblib\par
697             try:
698                 if getattr(self._backend, 'supports_timeout', False):
--> 699                     self._output.extend(job.get(timeout=self.timeout))
700                 else:
701                     self._output.extend(job.get())


c:\programdata\anaconda3\envs\tensorflow\lib\multiprocessing\pool.py in get(self, timeo
636
637     def get(self, timeout=None):
--> 638         self.wait(timeout)
639         if not self.ready():
640             raise TimeoutError


c:\programdata\anaconda3\envs\tensorflow\lib\multiprocessing\pool.py in wait(self, time
633
634     def wait(self, timeout=None):
--> 635         self._event.wait(timeout)
636
637     def get(self, timeout=None):
```

```
        c:\programdata\anaconda3\envs\tensorflow\lib\threading.py in wait(self, timeout)
        547             signaled = self._flag
        548             if not signaled:
    --> 549                 signaled = self._cond.wait(timeout)
        550             return signaled
        551


        c:\programdata\anaconda3\envs\tensorflow\lib\threading.py in wait(self, timeout)
        291         try:    # restore state no matter what (e.g., KeyboardInterrupt)
        292             if timeout is None:
    --> 293                 waiter.acquire()
        294                 gotit = True
        295             else:


        KeyboardInterrupt:


[ERROR] [14:53:42:tornado.general] Uncaught exception, closing connection.
Traceback (most recent call last):
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\eventloop\zmqstream.py"
    callback(*args, **kwargs)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\tornado\stack_context.py", l
    return fn(*args, **kwargs)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    return self.dispatch_shell(stream, msg)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    handler(stream, idents, msg)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    self._abort_queues()
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    self._abort_queue(stream)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    poller.poll(50)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\sugar\poll.py", line 99
    return zmq_poll(self.sockets, timeout=timeout)
  File "zmq\backend\cython\_poll.pyx", line 116, in zmq.backend.cython._poll.zmq_poll
  File "zmq\backend\cython\checkrc.pxd", line 12, in zmq.backend.cython.checkrc._check_rc
KeyboardInterrupt
[ERROR] [14:53:42:tornado.general] Uncaught exception, closing connection.
Traceback (most recent call last):
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\eventloop\zmqstream.py"
    self._handle_recv()
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\eventloop\zmqstream.py"
    self._run_callback(callback, msg)
```

```
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\eventloop\zmqstream.py"
    callback(*args, **kwargs)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\tornado\stack_context.py", l
    return fn(*args, **kwargs)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    return self.dispatch_shell(stream, msg)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    handler(stream, idents, msg)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    self._abort_queues()
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    self._abort_queue(stream)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\ipykernel\kernelbase.py", li
    poller.poll(50)
  File "c:\programdata\anaconda3\envs\tensorflow\lib\site-packages\zmq\sugar\poll.py", line 99
    return zmq_poll(self.sockets, timeout=timeout)
  File "zmq\backend\cython\_poll.pyx", line 116, in zmq.backend.cython._poll.zmq_poll
  File "zmq\backend\cython\checkrc.pxd", line 12, in zmq.backend.cython.checkrc._check_rc
KeyboardInterrupt
```

```
In [ ]: results = pd.DataFrame(grid_cv.cv_validation_scores)
        display(results)
```

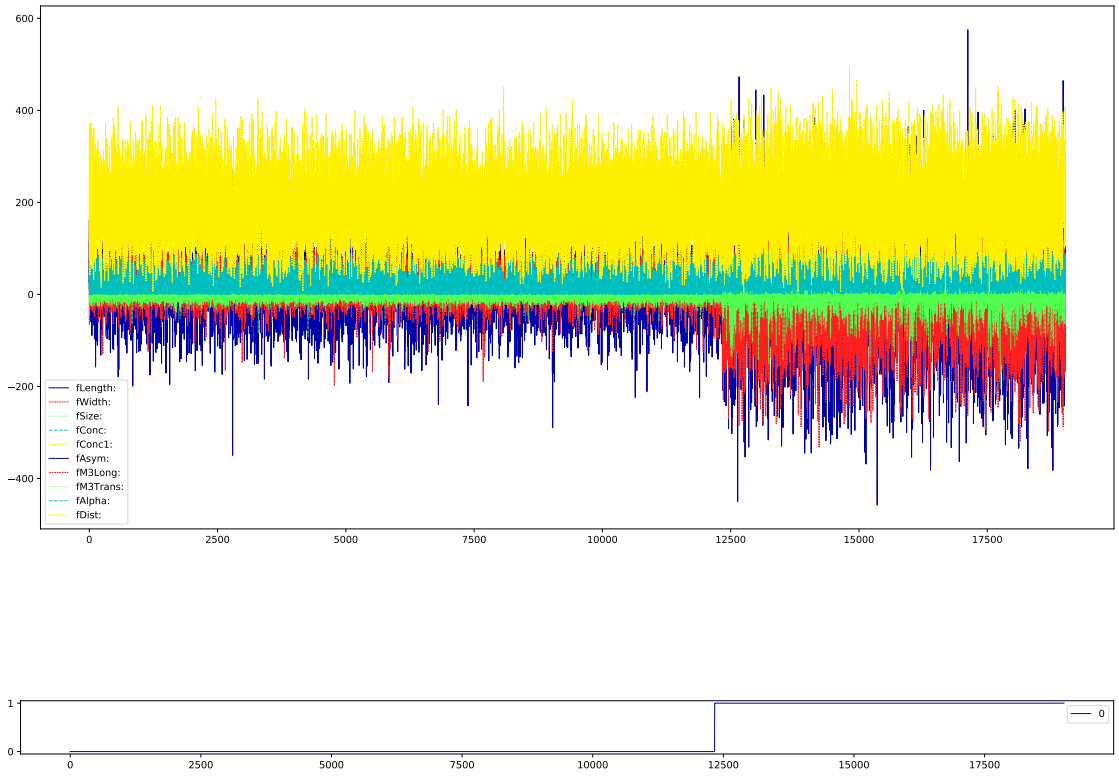
```
In [ ]: mglearn.tools.heatmap(results, xlabel='alpha', xticklabels=param_grid['alpha'],
                              ylabel='l1_ratio', yticklabels=param_grid['l1_ratio'], cmap="viri
```

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 [3]: # 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 [4]: # 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));
```

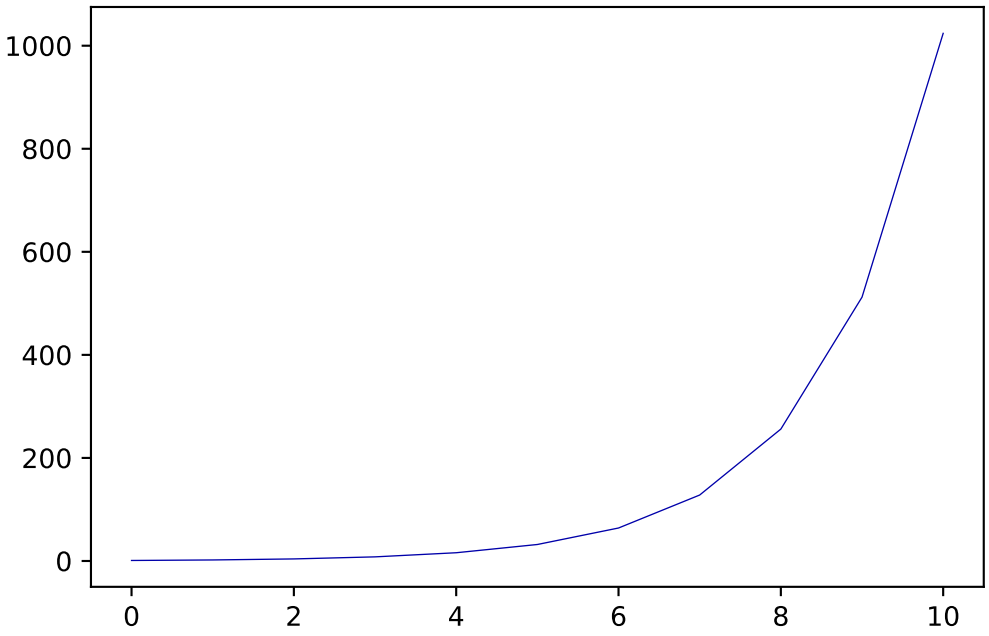


1 . Do a bias-variance analysis of both algorithms. For each, vary the number of trees on a log scale from 1 to 1024, and plot the bias error (squared), variance, and total error (in one plot per algorithm). Interpret the results. Which error is highest for small ensembles, and which reduced most by each algorithm as you use a larger ensemble? When are both algorithms under- or over-fitting? 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 [5]: estimates = np.logspace(0,10,num=11,base=2.0,endpoint=True,dtype='int')
        print(estimates)
        plt.plot(estimates)
```

```
[   1    2    4    8   16   32   64  128  256  512 1024]
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x180e257fb00>]
```

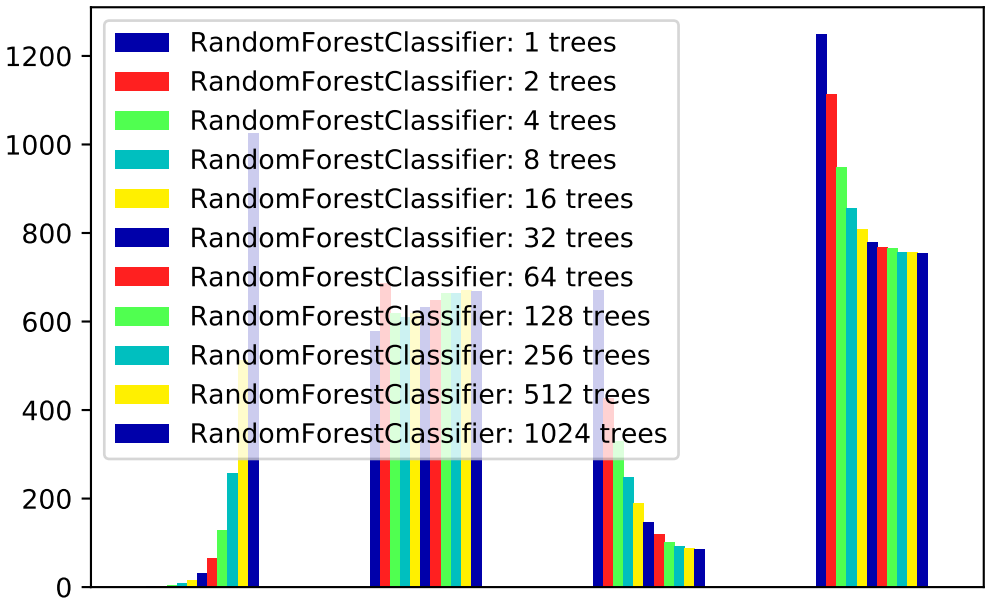


In [11]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.ensemble import RandomForestClassifier

scores = pd.DataFrame(columns=['Estimates', 'Bias squared', 'Variance', 'Total error']

def calc_print_bias_variance(y_all_pred, model_name, e):
    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_
               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" % (bias_sq, var, er
    scores.loc[model_name] = [e, bias_sq, var, error]

for estimate in estimates:
    print("Analyzing RandomForestClassifier with n_estimators: " + str(estimate))

    forest = RandomForestClassifier(n_estimators = estimate, random_state=2, n_jobs =

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

    # Bootstraps
    n_repeat = 100
    shuffle_split = ShuffleSplit(test_size=0.33, n_splits=n_repeat)
```

```
            # Train classifier on each bootstrap and score predictions
            for i, (train_index, test_index) in enumerate(shuffle_split.split(X)):
                X_train, X_test = X[train_index], X[test_index]
                y_train, y_test = y[train_index], y[test_index]

                # RandomForest
                forest.fit(X_train, y_train)
                y_forest_pred = forest.predict(X_test)

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

            pd.to_pickle(forest_all_pred, 'RF_Predictions_' + str(estimate) + '.pkl')

            calc_print_bias_variance(forest_all_pred, 'RandomForestClassifier: ' + str(estima

        # Showing the results
        display(scores)
        scores = scores.T
        ax = scores.plot(kind='bar')
        ax.set_xticks([])
        plt.legend(loc='best');
```

```
Analyzing RandomForestClassifier with n_estimators: 1
Bias squared: 577.27, Variance: 670.36, Total error: 1247.63
Analyzing RandomForestClassifier with n_estimators: 2
Bias squared: 686.32, Variance: 426.32, Total error: 1112.64
Analyzing RandomForestClassifier with n_estimators: 4
Bias squared: 618.57, Variance: 330.14, Total error: 948.71
Analyzing RandomForestClassifier with n_estimators: 8
Bias squared: 608.79, Variance: 247.19, Total error: 855.98
Analyzing RandomForestClassifier with n_estimators: 16
Bias squared: 619.26, Variance: 188.41, Total error: 807.67
Analyzing RandomForestClassifier with n_estimators: 32
Bias squared: 632.17, Variance: 146.47, Total error: 778.64
Analyzing RandomForestClassifier with n_estimators: 64
Bias squared: 648.64, Variance: 118.89, Total error: 767.53
Analyzing RandomForestClassifier with n_estimators: 128
Bias squared: 662.88, Variance: 101.88, Total error: 764.76
Analyzing RandomForestClassifier with n_estimators: 256
Bias squared: 663.48, Variance: 92.26, Total error: 755.74
Analyzing RandomForestClassifier with n_estimators: 512
Bias squared: 670.23, Variance: 86.63, Total error: 756.86
Analyzing RandomForestClassifier with n_estimators: 1024
Bias squared: 668.71, Variance: 85.07, Total error: 753.78
```

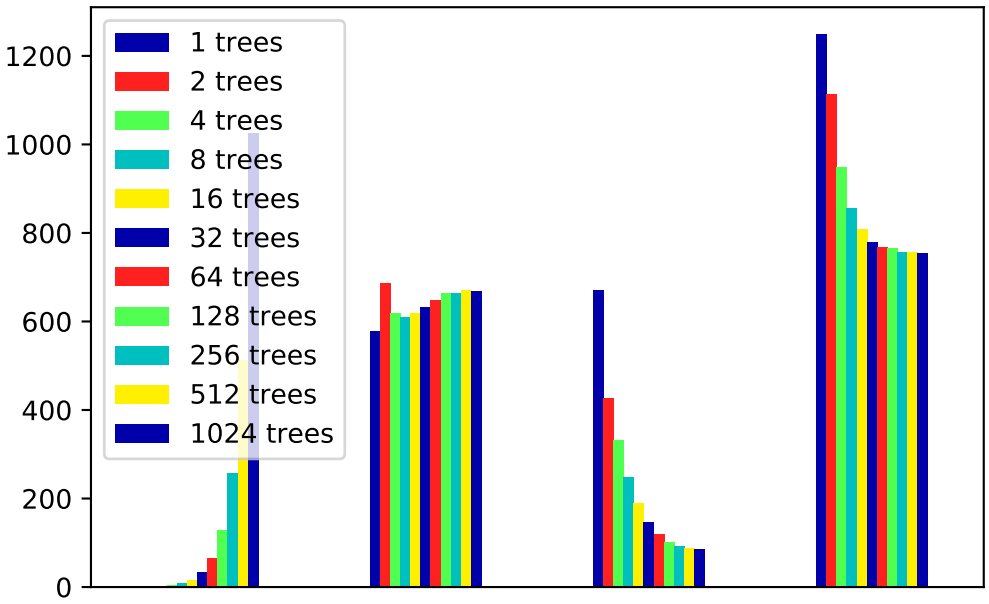
```
                                  Estimates  Bias squared  Variance  \
RandomForestClassifier: 1 trees         1.0        577.27    670.36
RandomForestClassifier: 2 trees         2.0        686.32    426.32
RandomForestClassifier: 4 trees         4.0        618.57    330.14
RandomForestClassifier: 8 trees         8.0        608.79    247.19
RandomForestClassifier: 16 trees       16.0        619.26    188.41
RandomForestClassifier: 32 trees       32.0        632.17    146.47
RandomForestClassifier: 64 trees       64.0        648.64    118.89
RandomForestClassifier: 128 trees     128.0        662.88    101.88
RandomForestClassifier: 256 trees     256.0        663.48     92.26
RandomForestClassifier: 512 trees     512.0        670.23     86.63
RandomForestClassifier: 1024 trees   1024.0        668.71     85.07

                                  Total error
RandomForestClassifier: 1 trees       1247.63
RandomForestClassifier: 2 trees       1112.64
RandomForestClassifier: 4 trees        948.71
RandomForestClassifier: 8 trees        855.98
RandomForestClassifier: 16 trees       807.67
RandomForestClassifier: 32 trees       778.64
RandomForestClassifier: 64 trees       767.53
RandomForestClassifier: 128 trees      764.76
RandomForestClassifier: 256 trees      755.74
RandomForestClassifier: 512 trees      756.86
RandomForestClassifier: 1024 trees     753.78
```



In [12]:
```
# Better graphs for RandomForest
scores = pd.DataFrame(columns=['Estimates', 'Bias squared', 'Variance', 'Total error']

for estimate in estimates:
    forest_all_pred = pd.read_pickle('RF_Predictions_' + str(estimate) + '.pkl')
    calc_print_bias_variance(forest_all_pred, str(estimate) + ' trees', estimate)

# Showing the results
display(scores)
scores = scores.T
ax = scores.plot(kind='bar')
ax.set_xticks([])
plt.legend(loc='best');
```

```
Bias squared: 577.27, Variance: 670.36, Total error: 1247.63
Bias squared: 686.32, Variance: 426.32, Total error: 1112.64
Bias squared: 618.57, Variance: 330.14, Total error: 948.71
Bias squared: 608.79, Variance: 247.19, Total error: 855.98
Bias squared: 619.26, Variance: 188.41, Total error: 807.67
Bias squared: 632.17, Variance: 146.47, Total error: 778.64
Bias squared: 648.64, Variance: 118.89, Total error: 767.53
Bias squared: 662.88, Variance: 101.88, Total error: 764.76
Bias squared: 663.48, Variance: 92.26, Total error: 755.74
Bias squared: 670.23, Variance: 86.63, Total error: 756.86
Bias squared: 668.71, Variance: 85.07, Total error: 753.78
```

	Estimates	Bias squared	Variance	Total error
1 trees	1.0	577.27	670.36	1247.63
2 trees	2.0	686.32	426.32	1112.64
4 trees	4.0	618.57	330.14	948.71
8 trees	8.0	608.79	247.19	855.98
16 trees	16.0	619.26	188.41	807.67
32 trees	32.0	632.17	146.47	778.64
64 trees	64.0	648.64	118.89	767.53
128 trees	128.0	662.88	101.88	764.76
256 trees	256.0	663.48	92.26	755.74
512 trees	512.0	670.23	86.63	756.86
1024 trees	1024.0	668.71	85.07	753.78



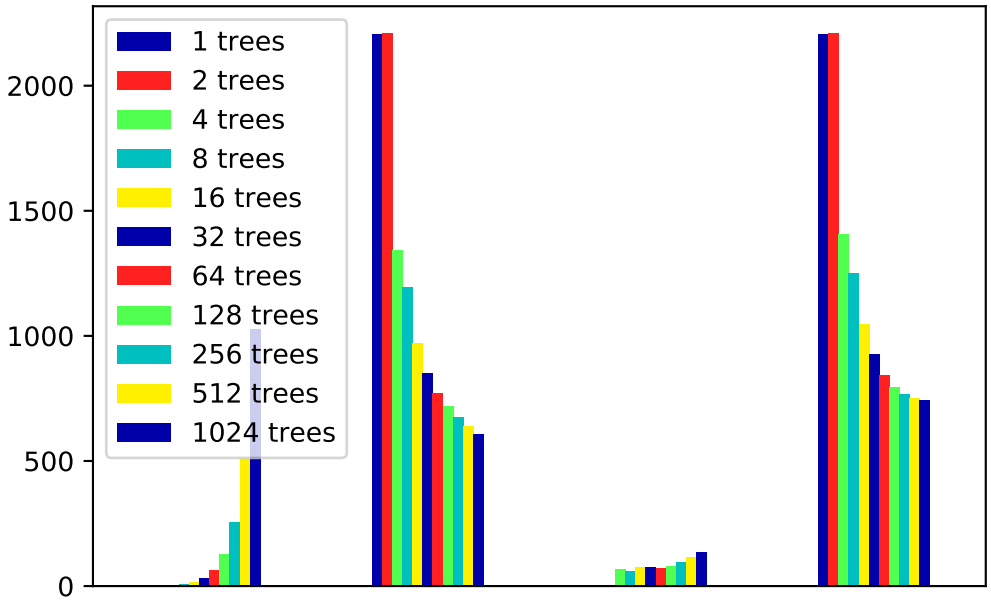

```
In [7]: from sklearn.ensemble import GradientBoostingClassifier

        scores = pd.DataFrame(columns=['Estimates', 'Bias squared', 'Variance', 'Total error'])

        for estimate in estimates:
            print("Analyzing GradientBoostingClassifier with n_estimators: " + str(estimate))

            gradient = GradientBoostingClassifier(n_estimators = estimate, random_state=2)

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

            # Bootstraps
            n_repeat = 100
            shuffle_split = ShuffleSplit(test_size=0.33, n_splits=n_repeat)

            # Train classifier on each bootstrap and score predictions
            for i, (train_index, test_index) in enumerate(shuffle_split.split(X)):
                X_train, X_test = X[train_index], X[test_index]
                y_train, y_test = y[train_index], y[test_index]

                # RandomForest
                gradient.fit(X_train, y_train)
                y_gradient_pred = gradient.predict(X_test)

                # Store predictions
```

```
                for i,index in enumerate(test_index):
                    gradient_all_pred[index].append(y_gradient_pred[i])

            pd.to_pickle(gradient_all_pred, 'GB_Predictions_' + str(estimate) + '.pkl')

            calc_print_bias_variance(gradient_all_pred, str(estimate) + ' trees', estimate)

        # Showing the results
        display(scores)
        scores = scores.T
        ax = scores.plot(kind='bar')
        ax.set_xticks([])
        plt.legend(loc='best');
```

```
Analyzing GradientBoostingClassifier with n_estimators: 1
Bias squared: 2206.53, Variance: 0.00, Total error: 2206.53
Analyzing GradientBoostingClassifier with n_estimators: 2
Bias squared: 2207.24, Variance: 0.00, Total error: 2207.24
Analyzing GradientBoostingClassifier with n_estimators: 4
Bias squared: 1341.99, Variance: 65.40, Total error: 1407.39
Analyzing GradientBoostingClassifier with n_estimators: 8
Bias squared: 1193.09, Variance: 58.56, Total error: 1251.65
Analyzing GradientBoostingClassifier with n_estimators: 16
Bias squared: 969.37, Variance: 75.10, Total error: 1044.47
Analyzing GradientBoostingClassifier with n_estimators: 32
Bias squared: 851.61, Variance: 73.20, Total error: 924.81
Analyzing GradientBoostingClassifier with n_estimators: 64
Bias squared: 770.84, Variance: 70.41, Total error: 841.25
Analyzing GradientBoostingClassifier with n_estimators: 128
Bias squared: 717.04, Variance: 78.77, Total error: 795.81
Analyzing GradientBoostingClassifier with n_estimators: 256
Bias squared: 673.39, Variance: 93.65, Total error: 767.04
Analyzing GradientBoostingClassifier with n_estimators: 512
Bias squared: 636.93, Variance: 113.22, Total error: 750.15
Analyzing GradientBoostingClassifier with n_estimators: 1024
Bias squared: 607.62, Variance: 135.62, Total error: 743.24
```

	Estimates	Bias squared	Variance	Total error
1 trees	1.0	2206.53	0.00	2206.53
2 trees	2.0	2207.24	0.00	2207.24
4 trees	4.0	1341.99	65.40	1407.39
8 trees	8.0	1193.09	58.56	1251.65
16 trees	16.0	969.37	75.10	1044.47
32 trees	32.0	851.61	73.20	924.81
64 trees	64.0	770.84	70.41	841.25
128 trees	128.0	717.04	78.77	795.81
256 trees	256.0	673.39	93.65	767.04

```
512 trees       512.0        636.93    113.22       750.15
1024 trees     1024.0        607.62    135.62       743.24
```



The first thing that we noticed from the graphs above is that RandomForest's bias squared is increased the more trees are used, but GradientBoosting's bias squared is actually reduced the more trees are used. This means that the more trees are used in RandomForest, the more likely it is to underfit. However, the more trees are used in GradientBoosting, the less likely it is to underfit.

Variance is also reduced for RandomForest models with more trees, while the opposite is true for GradientBoosting: the more trees are used, the more variances there are. This means that the more trees are used in RandomForest, the more likely it is to overfit. In the mean time, the more trees are used in GradientBoosting, the less likely it is to overfit.

On another hand, RandomForest's total error is reduced the more trees are used. The same also happens with GradientBoosting. This means that the more trees are used, the more accurate the results of these models are.

In general, it seems that RandomForest performs the best with small ensemble, while GradientBoosting performs better with more ensembles. This is not unexpected considering the way RandomForest works: the more trees it creates, the better the algorithm can locate and classify outliers. This gives more accurate results, but at the same time it also results in more chance of overfitting, which was proven by our bias-variance-error experiment.

On the other hand, GradientBoosting works by increasing the weights of the badly predicted points with each iteration. Thus, the more trees are used, the more iterations there are, and the more accurate the results will be. Furthermore, because the wrong predictions from the previous iterations were flagged, the less likely it will overfit, making the results better in general. This was also proven in our bias-variance-error experiment.

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 [36]:
```
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 [39]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)

gradient_01_1 = GradientBoostingClassifier(n_estimators = 500, learning_rate=0.1, max
gradient_01_4 = GradientBoostingClassifier(n_estimators = 500, learning_rate=0.1, max
gradient_1_1 = GradientBoostingClassifier(n_estimators = 500, learning_rate=1, max_de
```

```
        gradient_1_4 = GradientBoostingClassifier(n_estimators = 500, learning_rate=1, max_dep

        gradient_01_1.fit(X_train, y_train)
        pd.to_pickle(gradient_01_1, 'gradient_01_4' + str(iteration) + '.pkl')

        gradient_01_4.fit(X_train, y_train)
        pd.to_pickle(gradient_01_4, 'gradient_01_4' + str(iteration) + '.pkl')

        gradient_1_1.fit(X_train, y_train)
        pd.to_pickle(gradient_1_1, 'gradient_1_1' + str(iteration) + '.pkl')

        gradient_1_4.fit(X_train, y_train)
        pd.to_pickle(gradient_1_4, 'gradient_1_4' + str(iteration) + '.pkl')

        plt.rcParams['figure.dpi'] = 100
        validation_curve([gradient_01_1, gradient_01_4, gradient_1_1, gradient_1_4], X_test, y
```

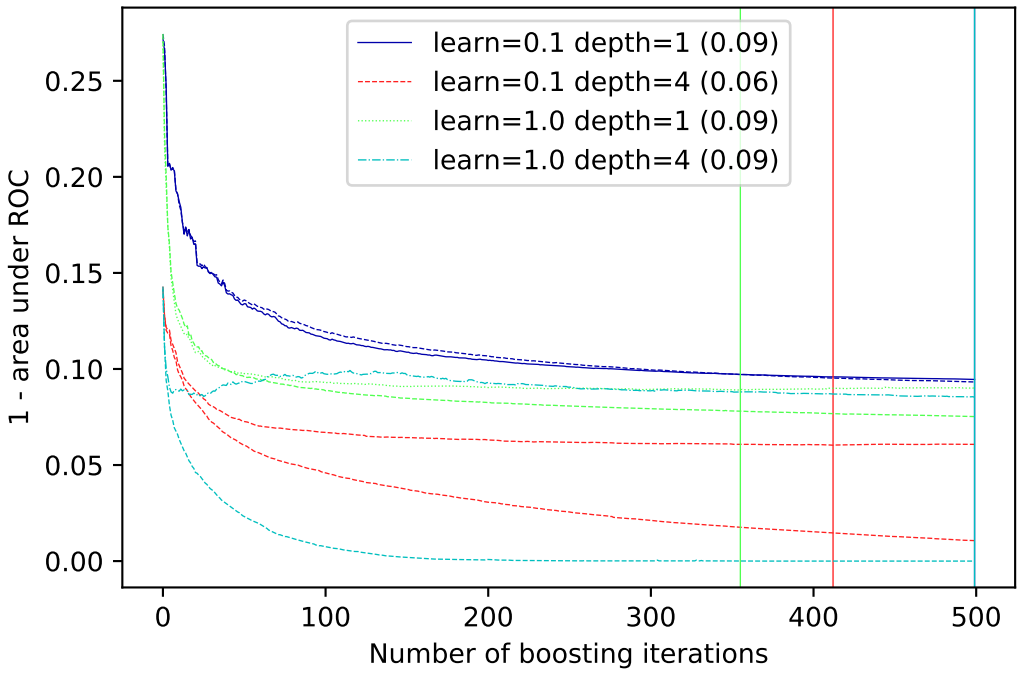
```
Out[39]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                       learning_rate=0.1, loss='deviance', max_depth=1,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500,
                       presort='auto', random_state=None, subsample=1.0, verbose=0,
                       warm_start=False)
```

```
Out[39]: 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=500,
                       presort='auto', random_state=None, subsample=1.0, verbose=0,
                       warm_start=False)
```

```
Out[39]: 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=500,
                       presort='auto', random_state=None, subsample=1.0, verbose=0,
                       warm_start=False)
```

```
Out[39]: 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=500,
presort='auto', random_state=None, subsample=1.0, verbose=0,
warm_start=False)
```



From the validation curves above, we can immediately see that there are a couple of configurations where the AUROC scores of the test and training data are widely different, namely the classifiers with tree depth 4 (red and light blue). Because there is such significant difference in the test and training scores for the models with tree depth 4, we suspect that these models are under- or overfitting.

Aside from the depth, the learning rate also seems to give influence to under- or overfit as well. If we look at the classifiers with tree depth 1 (dark blue and green), it seems that the model with learning rate 1 (green) under- or overfits more than the model with learning rate 0.1 (dark blue). This is understandable because the higher the learning rate, the quicker GradientBoosting learns about the data and the more likely it fits (and then overfits) the data. By slowing the learning of GradientBoosting classifier, we prevent the model from overfitting.

Even though more tree depths seems to result in more under- or overfitting, we can also see that the scores of models with tree depth 4 (red and light blue) are better compared to the models with less depth, signifying better accuracy. This makes sense as having more trees allows a classifier to locate and classify more points in the data.

Thus, it would seem that combining low learning rate with high tree depth (red curve) seems to be the most optimal choice, since it offers more accuracy and at the same time lowers the chance of under- or overfitting.