# Assignment 02a_Linear_regression_regul_poly

October 8, 2019

## 0.1 Assignment 02a - Linear Regression, Regularization and Polynomial Regression

Group:

- Jannik Bucher
- Dennis Imhof

### 0.1.1 Using dataset: SkillCraft1 Master Table Dataset

SkillCraft1 on UCI Machine Learning Repository

**Notes:**

- Typically "LeagueIndex" is used as the response variable. Since "LeagueIndex" is a categorical (ordinal) variable and we are supposed to perform linear regression, we will predict APM (actions per minute) from the most promising variables.
- Also, we will split the data into train and test set and perform cross-validation on the training data instead of splitting the training data into fixed train/validation sets.

### 0.1.2 Assignments

Before performing the practical work, you need download the dataset accordingly to the option on your machine (or cloud service) 1. Write a program that splits the original sample into a training set and a test set (training set, validation set, test set) with train_test_split method of Skikit Learn library

2. Using the scikit-learn library (http://scikit-learn.org/stable/), train the linear regression model for the training sample (example: http://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py)
3. Check the accuracy of the model from the test set
4. Build a model using a polynomial function (example: http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html#sphx-glr-auto-examples-model-selection-plot-underfitting-overfitting-py). Build plots with the dependence of the error on the degree of the polynomial function.

5. Build a model using regularization (example: http://scikit-learn.org/stable/modules/linear_model.html). On the basis of experiments, select parameters for regularization. Build plots with the dependence of the error on the regularization coefficient.

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import cross_val_predict, cross_val_score,␣
↪GridSearchCV
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LassoCV, RidgeCV, Lasso, Ridge
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.feature_selection import SelectFromModel
```

[2]:
```
df = pd.read_csv("data/SkillCraft1_Dataset.csv")
```

### 0.1.3 EDA

[3]:
```
df.head()
```

[3]:
```
   GameID  LeagueIndex Age HoursPerWeek TotalHours       APM  SelectByHotkeys  \
0      52            5  27           10       3000  143.7180         0.003515
1      55            5  23           10       5000  129.2322         0.003304
2      56            4  30           10        200   69.9612         0.001101
3      57            3  19           20        400  107.6016         0.001034
4      58            3  32           10        500  122.8908         0.001136

   AssignToHotkeys  UniqueHotkeys  MinimapAttacks  MinimapRightClicks  \
0         0.000220              7        0.000110            0.000392
1         0.000259              4        0.000294            0.000432
2         0.000336              4        0.000294            0.000461
3         0.000213              1        0.000053            0.000543
4         0.000327              2        0.000000            0.001329

   NumberOfPACs  GapBetweenPACs  ActionLatency  ActionsInPAC  \
0      0.004849         32.6677        40.8673        4.7508
1      0.004307         32.9194        42.3454        4.8434
2      0.002926         44.6475        75.3548        4.0430
3      0.003783         29.2203        53.7352        4.9155
4      0.002368         22.6885        62.0813        9.3740
```

```
   TotalMapExplored  WorkersMade  UniqueUnitsMade  ComplexUnitsMade  \
0                28     0.001397                6               0.0
1                22     0.001194                5               0.0
2                22     0.000745                6               0.0
3                19     0.000426                7               0.0
4                15     0.001174                4               0.0

   ComplexAbilitiesUsed
0              0.000000
1              0.000208
2              0.000189
3              0.000384
4              0.000019
```

[4]:
```
df.describe()
```

[4]:
```
             GameID  LeagueIndex          APM  SelectByHotkeys  \
count   3395.000000  3395.000000  3395.000000      3395.000000
mean    4805.012371     4.184094   117.046947         0.004299
std     2719.944851     1.517327    51.945291         0.005284
min       52.000000     1.000000    22.059600         0.000000
25%     2464.500000     3.000000    79.900200         0.001258
50%     4874.000000     4.000000   108.010200         0.002500
75%     7108.500000     5.000000   142.790400         0.005133
max    10095.000000     8.000000   389.831400         0.043088

       AssignToHotkeys  UniqueHotkeys  MinimapAttacks  MinimapRightClicks  \
count      3395.000000    3395.000000     3395.000000         3395.000000
mean          0.000374       4.364654        0.000098            0.000387
std           0.000225       2.360333        0.000166            0.000377
min           0.000000       0.000000        0.000000            0.000000
25%           0.000204       3.000000        0.000000            0.000140
50%           0.000353       4.000000        0.000040            0.000281
75%           0.000499       6.000000        0.000119            0.000514
max           0.001752      10.000000        0.003019            0.004041

       NumberOfPACs  GapBetweenPACs  ActionLatency  ActionsInPAC  \
count   3395.000000     3395.000000    3395.000000   3395.000000
mean       0.003463       40.361562      63.739403      5.272988
std        0.000992       17.153570      19.238869      1.494835
min        0.000679        6.666700      24.093600      2.038900
25%        0.002754       28.957750      50.446600      4.272850
50%        0.003395       36.723500      60.931800      5.095500
75%        0.004027       48.290500      73.681300      6.033600
max        0.007971      237.142900     176.372100     18.558100

       TotalMapExplored  WorkersMade  UniqueUnitsMade  ComplexUnitsMade  \
```

```
count       3395.000000  3395.000000       3395.000000        3395.000000
mean          22.131664     0.001032          6.534021           0.000059
std            7.431719     0.000519          1.857697           0.000111
min            5.000000     0.000077          2.000000           0.000000
25%           17.000000     0.000683          5.000000           0.000000
50%           22.000000     0.000905          6.000000           0.000000
75%           27.000000     0.001259          8.000000           0.000086
max           58.000000     0.005149         13.000000           0.000902

       ComplexAbilitiesUsed
count           3395.000000
mean               0.000142
std                0.000265
min                0.000000
25%                0.000000
50%                0.000020
75%                0.000181
max                0.003084
```

[5]:
```
# No missing values in any of the columns
df.isnull().any()
```

[5]:
```
GameID                  False
LeagueIndex             False
Age                     False
HoursPerWeek            False
TotalHours              False
APM                     False
SelectByHotkeys         False
AssignToHotkeys         False
UniqueHotkeys           False
MinimapAttacks          False
MinimapRightClicks      False
NumberOfPACs            False
GapBetweenPACs          False
ActionLatency           False
ActionsInPAC            False
TotalMapExplored        False
WorkersMade             False
UniqueUnitsMade         False
ComplexUnitsMade        False
ComplexAbilitiesUsed    False
dtype: bool
```

[6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3395 entries, 0 to 3394
Data columns (total 20 columns):
```

```
GameID                   3395 non-null int64
LeagueIndex              3395 non-null int64
Age                      3395 non-null object
HoursPerWeek             3395 non-null object
TotalHours               3395 non-null object
APM                      3395 non-null float64
SelectByHotkeys          3395 non-null float64
AssignToHotkeys          3395 non-null float64
UniqueHotkeys            3395 non-null int64
MinimapAttacks           3395 non-null float64
MinimapRightClicks       3395 non-null float64
NumberOfPACs             3395 non-null float64
GapBetweenPACs           3395 non-null float64
ActionLatency            3395 non-null float64
ActionsInPAC             3395 non-null float64
TotalMapExplored         3395 non-null int64
WorkersMade              3395 non-null float64
UniqueUnitsMade          3395 non-null int64
ComplexUnitsMade         3395 non-null float64
ComplexAbilitiesUsed     3395 non-null float64
dtypes: float64(12), int64(5), object(3)
memory usage: 530.6+ KB
```

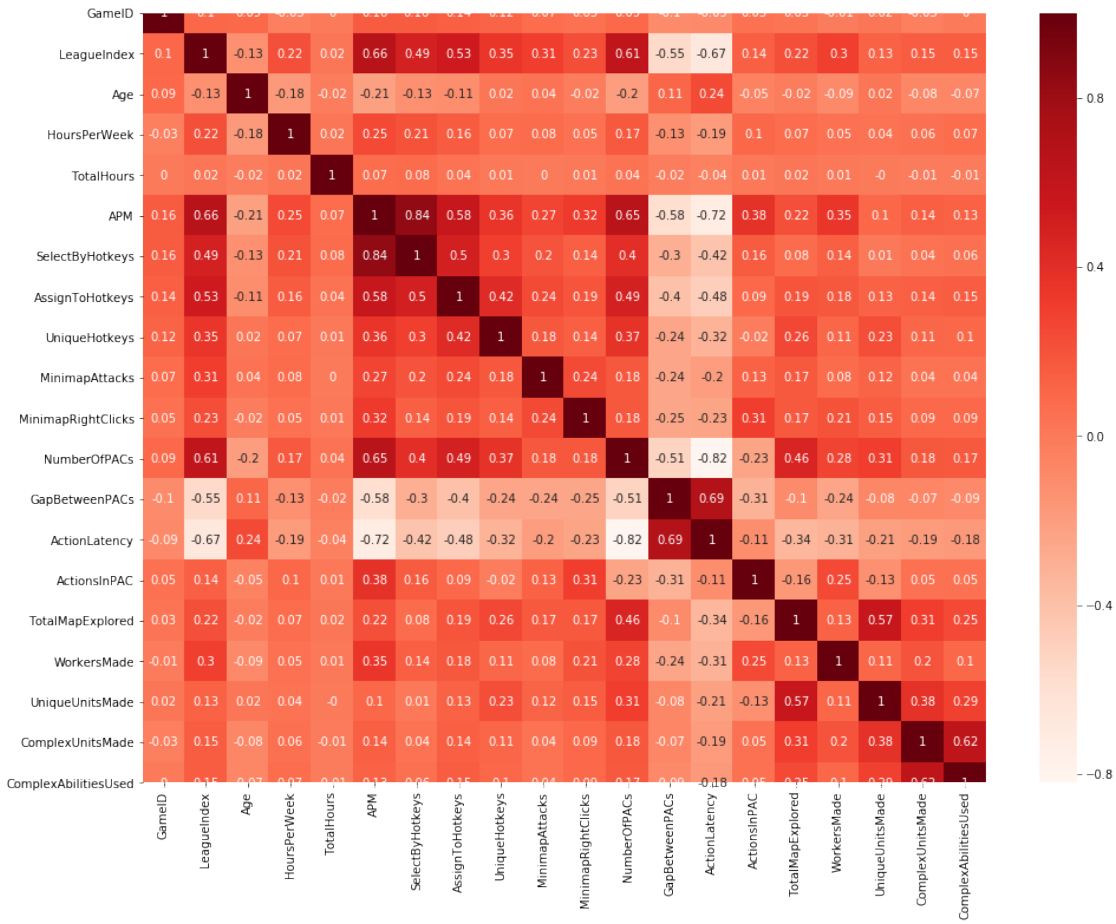
[7]:
```
# Though there were no None values in the dataset, closer inspection reveals␣
↪missing values marked with "?"
# Convert the object-variables to numeric and set the missing values to None
missing_features = ["Age", "HoursPerWeek", "TotalHours"]

for col in missing_features:
    df[col] = pd.to_numeric(df[col], errors="coerce")
```

[8]:
```
df.isna().sum()
```

[8]:
```
GameID                  0
LeagueIndex             0
Age                    55
HoursPerWeek           56
TotalHours             57
APM                     0
SelectByHotkeys         0
AssignToHotkeys         0
UniqueHotkeys           0
MinimapAttacks          0
MinimapRightClicks      0
NumberOfPACs            0
GapBetweenPACs          0
ActionLatency           0
ActionsInPAC            0
```

```
TotalMapExplored          0
WorkersMade               0
UniqueUnitsMade           0
ComplexUnitsMade          0
ComplexAbilitiesUsed      0
dtype: int64
```

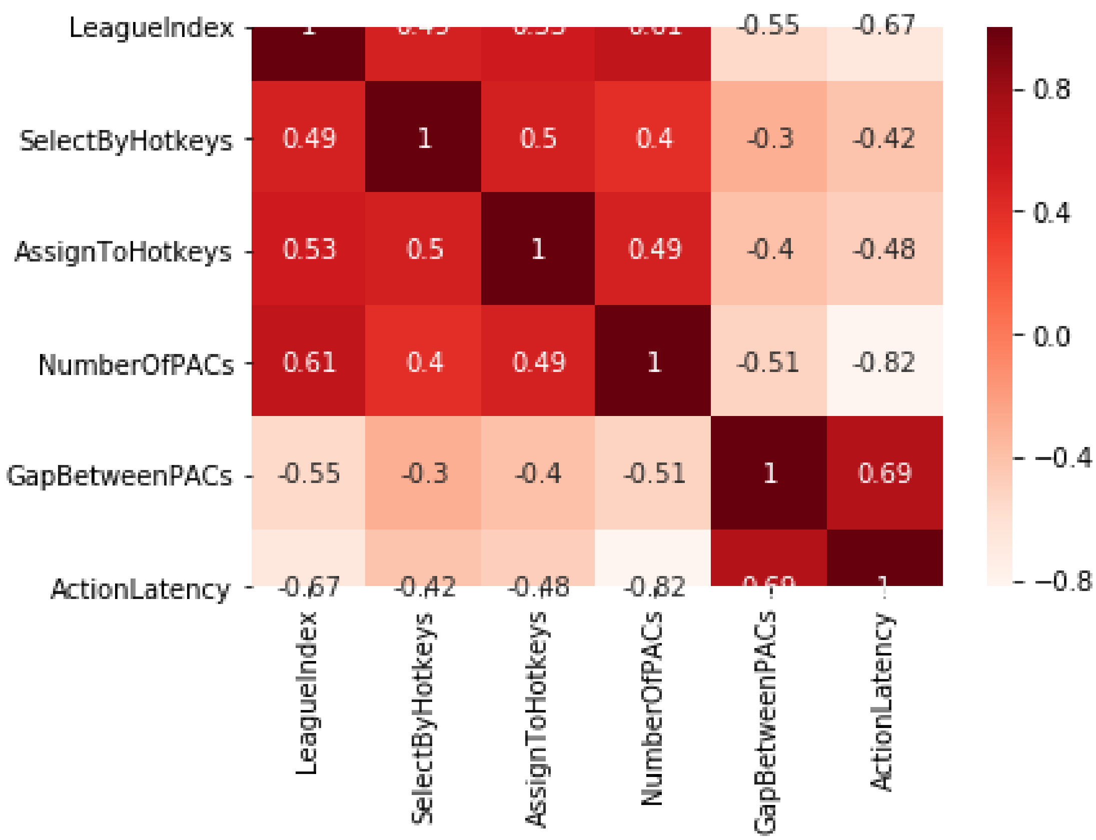
[9]:
```python
# Let's see if we find interessting patterns in a heatmap visualizing the correlation matrix
plt.figure(figsize=(16,12))
cor = df.corr().round(2)
sns.heatmap(cor, annot=True, cmap=plt.cm.Reds)
plt.show()
```




[10]:
```python
# Get the variables highly correlated with the response variable
cor_response = abs(cor["APM"])
relevant_features = cor_response[cor_response>0.5].drop("APM")
relevant_features
```

[10]:
```
LeagueIndex        0.66
SelectByHotkeys    0.84
AssignToHotkeys    0.58
NumberOfPACs       0.65
GapBetweenPACs     0.58
ActionLatency      0.72
Name: APM, dtype: float64
```

[11]:
```
# Visualize the correlation between the relevant features
corr_pred = df[relevant_features.index].corr().round(2)
sns.heatmap(corr_pred, annot=True, cmap=plt.cm.Reds)
plt.show()
```




### 0.1.4 Splitting the dataset into train/test

Before doing any modeling

[12]:
```
train_X, test_X, train_y, test_y = train_test_split(df.drop("APM",axis=1),␣
 ↪df["APM"], train_size=0.8, random_state=42)
```

### 0.1.5 Feature normalization

[13]:
```
remaining_features = train_X.columns.drop(missing_features)

numeric_transformer = Pipeline([('scale', StandardScaler())])
imputer = Pipeline([('impute', SimpleImputer(strategy='median')),
                    ('scale', numeric_transformer)])



preprocessor = ColumnTransformer(
    transformers=[
        ('imp', imputer, missing_features),
        ('num', numeric_transformer, remaining_features)])
```

[14]:
```
train_X = pd.DataFrame(preprocessor.fit_transform(train_X), columns=train_X.
 ↪columns)
```

[15]:
```
train_X.describe()
```

[15]:
```
             GameID   LeagueIndex            Age   HoursPerWeek     TotalHours  \
count  2.716000e+03  2.716000e+03   2.716000e+03   2.716000e+03   2.716000e+03
mean  -3.479462e-16 -4.055012e-17   2.616137e-18  -1.046455e-16  -8.633251e-17
std    1.000184e+00  1.000184e+00   1.000184e+00   1.000184e+00   1.000184e+00
min   -1.363021e+00 -1.315350e+00  -5.287988e-02  -1.779399e+00  -2.104066e+00
25%   -6.331381e-01 -6.502364e-01  -3.740366e-02  -8.486454e-01  -7.894483e-01
50%   -1.465498e-01 -3.176797e-01  -2.698197e-02   5.168607e-02  -1.321394e-01
75%    5.833327e-01  3.474335e-01  -1.134942e-02   8.476730e-01   5.251695e-01
max    5.449216e+00  1.265203e+01   5.205544e+01   1.945471e+00   2.497096e+00

       SelectByHotkeys  AssignToHotkeys  UniqueHotkeys  MinimapAttacks  \
count     2.716000e+03     2.716000e+03   2.716000e+03    2.716000e+03
mean      6.278728e-17    -7.128973e-17  -1.399633e-16   -4.709046e-17
std       1.000184e+00     1.000184e+00   1.000184e+00    1.000184e+00
min      -8.307951e-01    -1.677631e+00  -1.866150e+00   -5.806462e-01
25%      -5.818109e-01    -7.536533e-01  -5.950089e-01   -5.806462e-01
50%      -3.448266e-01    -9.823826e-02  -1.712952e-01   -3.473055e-01
75%       1.820376e-01     5.668545e-01   6.761323e-01    1.112131e-01
max       7.395625e+00     6.141420e+00   2.370987e+00    1.697499e+01

       MinimapRightClicks  NumberOfPACs  GapBetweenPACs  ActionLatency  \
count        2.716000e+03  2.716000e+03    2.716000e+03   2.716000e+03
mean        -1.491198e-16 -2.746944e-17    1.968643e-16   6.998166e-16
std          1.000184e+00  1.000184e+00    1.000184e+00   1.000184e+00
min         -1.025473e+00 -2.801613e+00   -1.917108e+00  -2.047678e+00
25%         -6.537659e-01 -7.191526e-01   -6.705181e-01  -6.927804e-01
50%         -2.739262e-01 -6.801996e-02   -2.138966e-01  -1.487731e-01
75%          3.344726e-01  5.768667e-01    4.429944e-01   5.251209e-01
max          9.483518e+00  4.487375e+00    1.121705e+01   5.845117e+00
```

```
       ActionsInPAC  TotalMapExplored   WorkersMade  UniqueUnitsMade  \
count  2.716000e+03      2.716000e+03  2.716000e+03     2.716000e+03
mean   8.698655e-17     -2.459169e-16  6.671149e-17     1.138019e-16
std    1.000184e+00      1.000184e+00  1.000184e+00     1.000184e+00
min   -2.132266e+00     -2.314673e+00 -1.825463e+00    -2.441983e+00
25%   -6.656240e-01     -7.050841e-01 -6.721640e-01    -8.337324e-01
50%   -1.135236e-01     -3.442205e-02 -2.398668e-01    -2.976488e-01
75%    5.183354e-01      6.362400e-01  4.334208e-01     7.745184e-01
max    8.977585e+00      4.794345e+00  7.785008e+00     3.454937e+00

       ComplexUnitsMade  ComplexAbilitiesUsed
count      2.716000e+03          2.716000e+03
mean      -1.052995e-16          3.270171e-18
std        1.000184e+00          1.000184e+00
min       -5.367249e-01         -5.320214e-01
25%       -5.367249e-01         -5.320214e-01
50%       -5.367249e-01         -4.547275e-01
75%        2.393433e-01          1.565407e-01
max        7.484828e+00          1.084133e+01
```

### 0.1.6 Backward Selection

In the correlation matrix above we can see that many of the promising features seem to be correlated. Thus we will use backward selection to weed out the features where our null-hypothesis

$$H_0 : \omega_i = 0$$

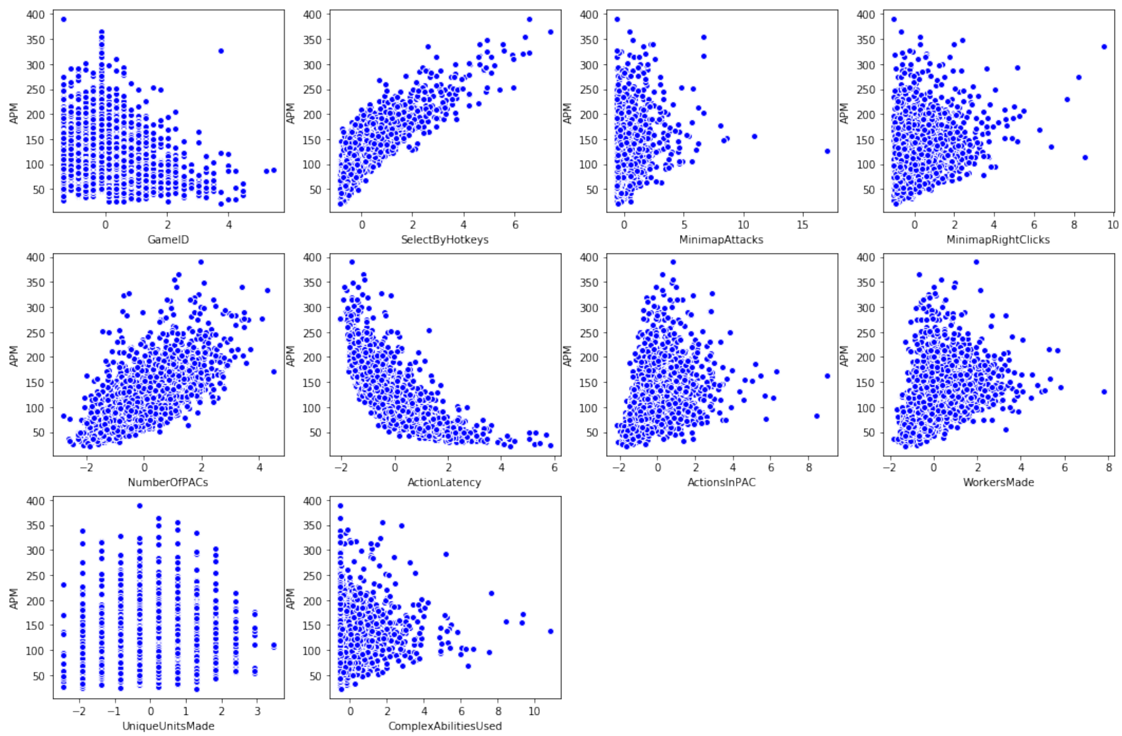
cannot be rejected, because of a p-value > 5%. Hereby $\omega_i$ is the slope estimate of the OLS linear regression.

[16]:
```python
import statsmodels.api as sm
# Perform backward selection and see which variables will remain with p-value <
 ↪5%

predictors = list(train_X.columns)
while len(predictors) > 0:
    X_1 = sm.add_constant(train_X[predictors].values)
    model = sm.OLS(train_y,X_1).fit()

    p_values = pd.Series(model.pvalues.values[1:], index = predictors)
    pmax = np.max(p_values)
    feature_with_p_max = p_values.idxmax()
    if(pmax>0.05):
        predictors.remove(feature_with_p_max)
    else:
        break
```

```python
print(predictors)
```

```
['GameID', 'SelectByHotkeys', 'MinimapAttacks', 'MinimapRightClicks',
'NumberOfPACs', 'ActionLatency', 'ActionsInPAC', 'WorkersMade',
'UniqueUnitsMade', 'ComplexAbilitiesUsed']
```

[17]:
```python
# Plot the distributions of the relevant variables vs the response variable

fig = plt.figure(figsize=(18,12))
for i,col in enumerate(predictors):
    ax = plt.subplot(np.ceil(len(predictors)/4), 4, i+1)
    ax.set_xlabel(col)
    ax.set_ylabel("APM")
    plt.plot(train_X[col], train_y, 'bo', mec='w')
#sns.pairplot(df_scaled, x_vars=predictors, y_vars=["APM"])
plt.show()
```




Judging from the pairplots of the response variable vs scaled features, it seems that

- SelectByHotkeys and NumberOfPACs are good choices for a linear fit
- ActionLatency might be a good choice for a polynomial fit

[18]:
```
def plot_regression(predictors, degrees, height=32):
    plt.figure(figsize=(25, height))
    j = 0
    for predictor in predictors:
        for i in range(len(degrees)):
            j += 1
            ax = plt.subplot(np.ceil(len(degrees)*len(predictors)/5), 5, j)

            polynomial_features = PolynomialFeatures(degree=degrees[i],
                                                     include_bias=False)
            linear_regression = LinearRegression()
            pipeline = Pipeline([("polynomial_features", polynomial_features),
                                 ("linear_regression", linear_regression)])
            pipeline.fit(train_X[[predictor]], train_y)

            # Evaluate the models using crossvalidation
            scores = cross_val_score(pipeline, train_X[[predictor]], train_y,
                                     scoring="neg_mean_squared_error", cv=10)

            xs = np.linspace(np.min(train_X[predictor]), np.
 ↪max(train_X[predictor]), 100)
            plt.plot(xs, pipeline.predict(xs.reshape(-1,1)), 'r-', label="Model")
            plt.scatter(train_X[[predictor]], train_y, edgecolor='b', s=20,␣
 ↪label="Samples")
            ax.set_xlabel(predictor)
            ax.set_ylabel("APM")
            plt.legend(loc="best")
            plt.title("Degree {}\nMSE = {:.4f} (+/- {:.2e})".format(degrees[i],␣
 ↪-scores.mean(), scores.std()))
    plt.show()
```

### 0.1.7 Linear Regression

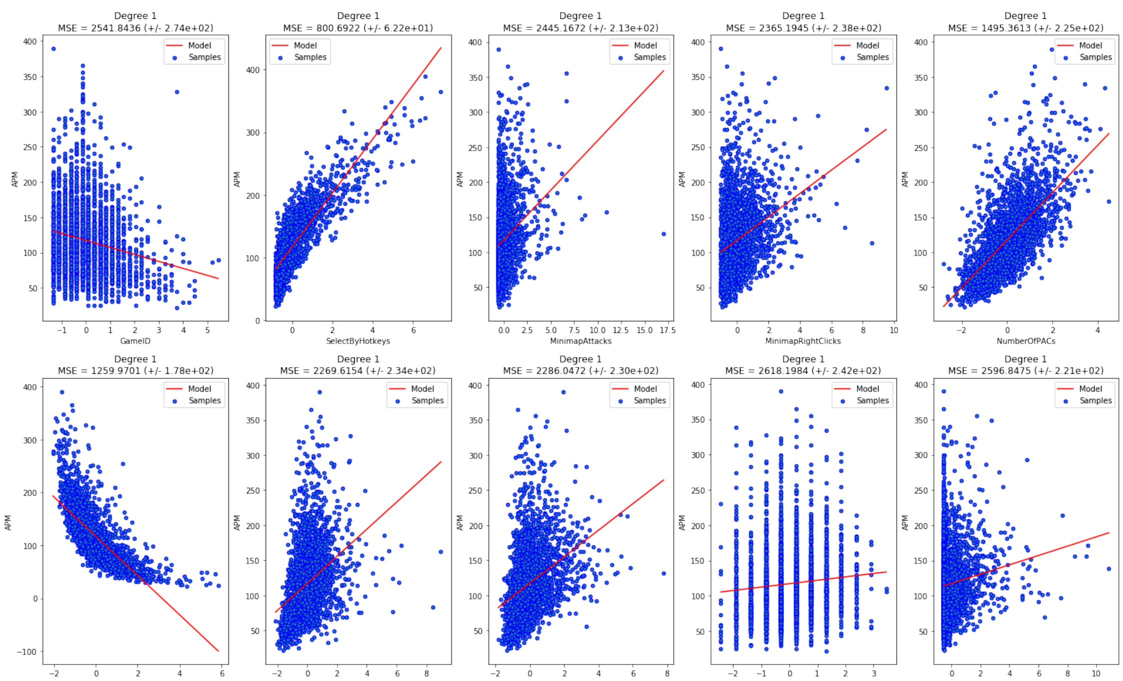
[19]:
```
errors = {}
coef = {}
intercept = {}

for predictor in predictors:
    lr = LinearRegression()
    errors[predictor] = np.mean(-cross_val_score(lr, train_X[[predictor]],␣
 ↪train_y, cv=10, scoring="neg_mean_squared_error"))

errors = pd.Series(list(errors.values()), index=errors.keys()).sort_values()
print("Mean Squared Errors (MSE)")
print(errors)
```

```
Mean Squared Errors (MSE)
```

```
SelectByHotkeys           800.692204
ActionLatency            1259.970131
NumberOfPACs             1495.361317
ActionsInPAC             2269.615395
WorkersMade              2286.047163
MinimapRightClicks       2365.194541
MinimapAttacks           2445.167153
GameID                   2541.843650
ComplexAbilitiesUsed     2596.847473
UniqueUnitsMade          2618.198413
dtype: float64
```

As expected by the pairplots SelectByHotkeys has the lowest MSE by far. Let's see the scatter-plots and fitted models.

[20]:
```python
plot_regression(predictors, [1], 15)
```




### 0.1.8 Polynomial Regression

Now let's see which features will perform best in a polynomial regression.

[21]:
```python
def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree), LinearRegression(**kwargs))

errors = {}
best_degree = {}
```

```
for predictor in predictors:
    poly_params = {'polynomialfeatures__degree': range(1,10)}
    poly_grid = GridSearchCV(PolynomialRegression(), poly_params,
                         cv=3, n_jobs=-1, scoring="neg_mean_squared_error")

    fit = poly_grid.fit(train_X[[predictor]], train_y)

    errors[predictor] = -fit.best_score_
    best_degree[predictor] = fit.best_params_["polynomialfeatures__degree"]
```

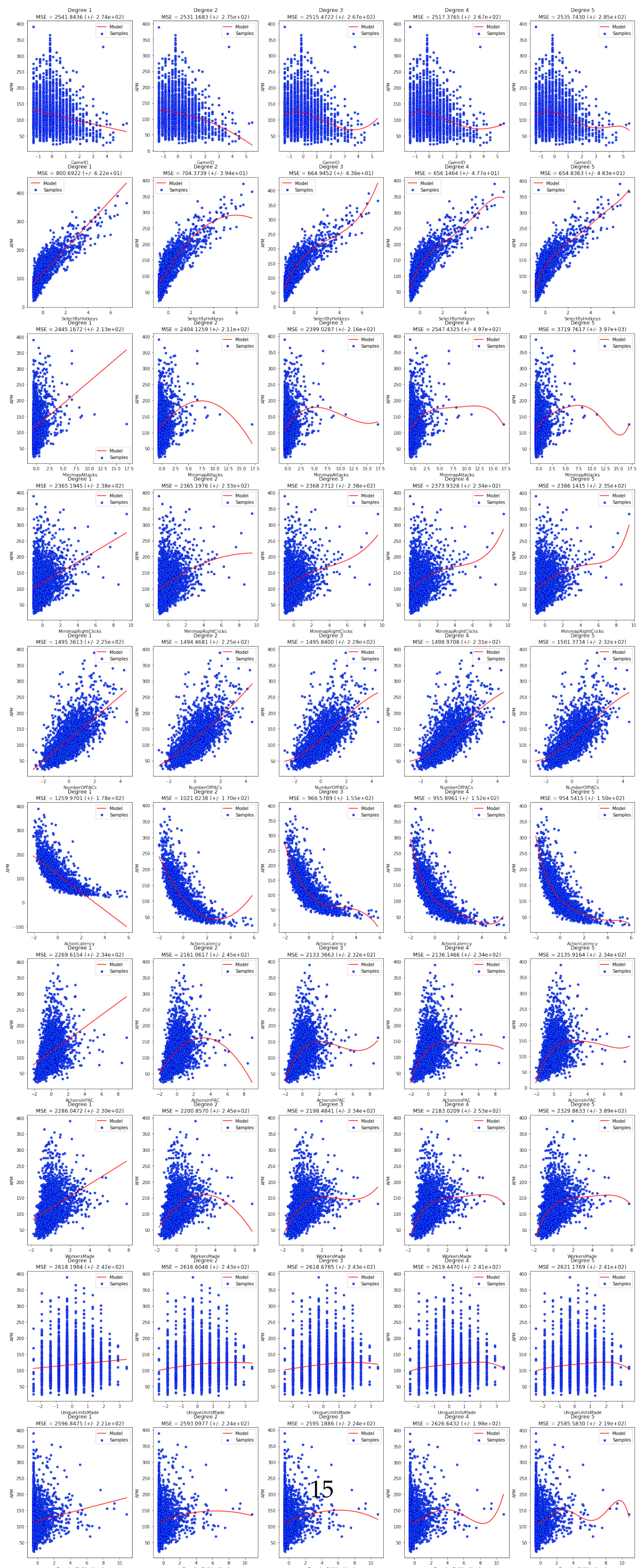
```
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[22]:
```
best_poly = pd.DataFrame(errors.values(), index=list(errors.keys()),␣
 ↪columns=["MSE"])
best_poly["Degree"] = best_degree.values()
best_poly.sort_values("MSE")
```

[22]:

| | MSE | Degree |
|---|---|---|
| SelectByHotkeys | 654.493598 | 4 |
| ActionLatency | 959.157374 | 5 |
| NumberOfPACs | 1492.644516 | 2 |
| ActionsInPAC | 2139.333205 | 3 |
| WorkersMade | 2182.349452 | 4 |
| MinimapRightClicks | 2359.703289 | 3 |
| MinimapAttacks | 2429.472090 | 2 |
| GameID | 2514.261008 | 3 |
| ComplexAbilitiesUsed | 2582.700209 | 5 |
| UniqueUnitsMade | 2619.616374 | 1 |

[23]:
```
plot_regression(predictors, [1, 2, 3, 4, 5], 65)
```

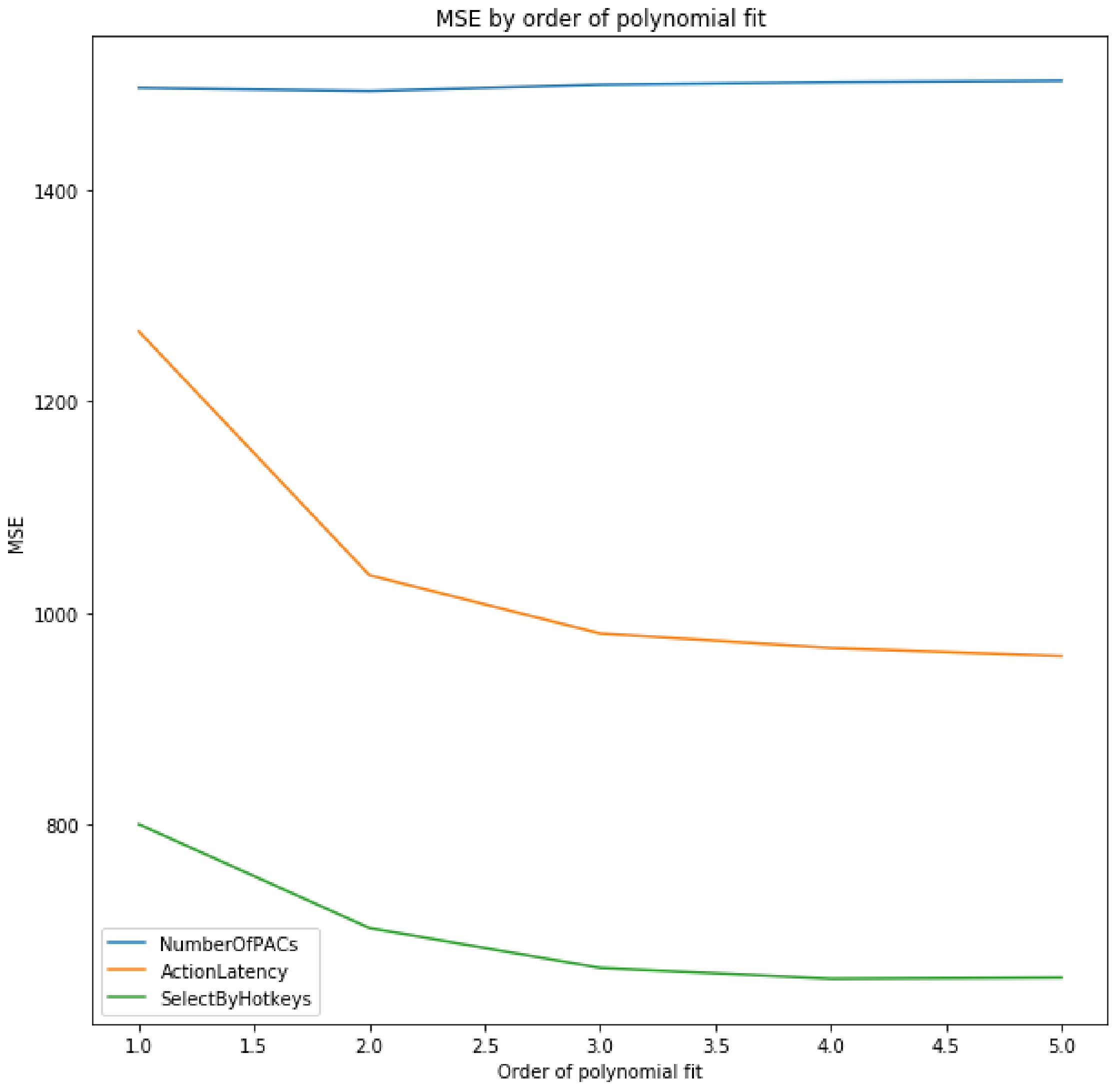

Degree 1
MSE = 2541.8436 (+/- 2.74e+02)
Degree 2
MSE = 2531.1683 (+/- 2.75e+02)
Degree 3
MSE = 2515.4722 (+/- 2.67e+02)
Degree 4
MSE = 2517.3765 (+/- 2.67e+02)
Degree 5
MSE = 2535.7430 (+/- 2.85e+02)
GameID
Degree 1
MSE = 800.6922 (+/- 6.22e+01)
Degree 2
MSE = 704.3739 (+/- 3.94e+01)
Degree 3
MSE = 664.9452 (+/- 4.36e+01)
Degree 4
MSE = 656.1464 (+/- 4.77e+01)
Degree 5
MSE = 654.8363 (+/- 4.83e+01)
SelectByHotkeys
Degree 1
MSE = 2445.1672 (+/- 2.13e+02)
Degree 2
MSE = 2404.1259 (+/- 2.11e+02)
Degree 3
MSE = 2399.0287 (+/- 2.16e+02)
Degree 4
MSE = 2547.4325 (+/- 4.97e+02)
Degree 5
MSE = 3719.7617 (+/- 3.97e+03)
MinimapAttacks
Degree 1
MSE = 2365.1945 (+/- 2.38e+02)
Degree 2
MSE = 2365.1976 (+/- 2.33e+02)
Degree 3
MSE = 2368.2712 (+/- 2.38e+02)
Degree 4
MSE = 2373.9328 (+/- 2.34e+02)
Degree 5
MSE = 2386.1415 (+/- 2.35e+02)
MinimapRightClicks
Degree 1
MSE = 1495.3613 (+/- 2.25e+02)
Degree 2
MSE = 1494.4681 (+/- 2.25e+02)
Degree 3
MSE = 1495.8400 (+/- 2.29e+02)
Degree 4
MSE = 1498.9708 (+/- 2.31e+02)
Degree 5
MSE = 1501.3734 (+/- 2.32e+02)
NumberOfPACs
Degree 1
MSE = 1259.9701 (+/- 1.78e+02)
Degree 2
MSE = 1021.0238 (+/- 1.70e+02)
Degree 3
MSE = 966.5789 (+/- 1.55e+02)
Degree 4
MSE = 955.8961 (+/- 1.52e+02)
Degree 5
MSE = 954.5415 (+/- 1.50e+02)
ActionLatency
Degree 1
MSE = 2269.6154 (+/- 2.34e+02)
Degree 2
MSE = 2161.0617 (+/- 2.45e+02)
Degree 3
MSE = 2133.3663 (+/- 2.32e+02)
Degree 4
MSE = 2136.1466 (+/- 2.34e+02)
Degree 5
MSE = 2135.9164 (+/- 2.34e+02)
ActionsInPAC
Degree 1
MSE = 2286.0472 (+/- 2.30e+02)
Degree 2
MSE = 2200.8570 (+/- 2.45e+02)
Degree 3
MSE = 2198.4841 (+/- 2.34e+02)
Degree 4
MSE = 2183.0209 (+/- 2.53e+02)
Degree 5
MSE = 2329.8633 (+/- 3.89e+02)
WorkersMade
Degree 1
MSE = 2618.1984 (+/- 2.42e+02)
Degree 2
MSE = 2616.6048 (+/- 2.43e+02)
Degree 3
MSE = 2618.6785 (+/- 2.43e+02)
Degree 4
MSE = 2619.4470 (+/- 2.41e+02)
Degree 5
MSE = 2621.1769 (+/- 2.41e+02)
UniqueUnitsMade
Degree 1
MSE = 2596.8475 (+/- 2.21e+02)
Degree 2
MSE = 2593.0977 (+/- 2.24e+02)
Degree 3
MSE = 2595.1886 (+/- 2.24e+02)
Degree 4
MSE = 2626.6432 (+/- 1.98e+02)
Degree 5
MSE = 2585.5830 (+/- 2.19e+02)
ComplexAbilitiesUsed
APM
Model
Samples

### 0.1.9 MSE vs. degree of polynomial fit

Here we'll only take a look at the three features `NumberOfPACs`, `ActionLatency` and `SelectByHotkeys` for which cross-validation has resulted in the lowest MSE.

[24]:
```
coefficients = {}
intercepts = {}
errors = {}

xs = np.arange(1,10,1)

poly_predictors = ["NumberOfPACs", "ActionLatency", "SelectByHotkeys"]
#poly_predictors = predictors
for predictor in poly_predictors:
    coefficients[predictor] = {}
    intercepts[predictor] = {}
    errors[predictor] = {}
    for i in range(5):
        linear_regression = LinearRegression()
        polynomial_features = PolynomialFeatures(degree=i+1,
                                                 include_bias=False)

        pipeline = Pipeline([("polynomial_features", polynomial_features),
                             ("linear_regression", linear_regression)])

        pipeline.fit(train_X[[predictor]], train_y)

        # Evaluate the models using crossvalidation
        errors[predictor][i+1] = -np.mean(cross_val_score(pipeline,␣
↪train_X[[predictor]], train_y,
                                 scoring="neg_mean_squared_error", cv=3))


errors = pd.DataFrame(errors)
print("Mean Squared Errors (MSE)")

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(title="MSE by order of polynomial fit")
plt.plot(errors)
ax.set_xlabel("Order of polynomial fit")
ax.set_ylabel("MSE")
plt.legend(labels=errors.columns )
plt.show()
```

```
Mean Squared Errors (MSE)
```




#### 0.1.10 Multivariate Polynomial Fit with the features selected by backward selection

[25]:
```
errors = {}
for i in range(1,6):
    pr = PolynomialRegression(degree=i)
    mse = -np.mean(cross_val_score(pr, train_X[predictors], train_y, cv=3,␣
 ↪scoring="neg_mean_squared_error"))
    errors[i] = np.round(mse,2)
```

[26]:
```
pd.Series(errors).sort_values()
```

[26]:
```
2     9.11
3    20.36
1    67.72
```

```
4       4672.71
5    5767093.73
dtype: float64
```

We found a MSE of 9.11 (!) for a quadratic fit of the model utilizing the features selected by backward selection.

## 0.2 Regularization

### 0.2.1 The Lasso

[27]:
```python
lasso_alphas = 10**np.linspace(2, -3, 100)

parameters = {'alpha' : lasso_alphas}
lasso_grid = GridSearchCV(Lasso(random_state=42), param_grid=parameters, cv=10,␣
↪n_jobs=-1, scoring='neg_mean_squared_error')
lasso_grid.fit(train_X, train_y)
```

```
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[27]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=1000, normalize=False, positive=False,
                             precompute=False, random_state=42,
                             selection='cyclic', tol=0.0001, warm_start=False),
             iid='warn', n_jobs=-1,
             param_grid={'alpha': array([1.00000000e+02, 8.90215085e+01,
7.92482898e+01, 7.05480231e+01,
       6.280291...
       9.11162756e-03, 8.11130831e-03, 7.22080902e-03, 6.42807312e-03,
       5.72236766e-03, 5.09413801e-03, 4.53487851e-03, 4.03701726e-03,
       3.59381366e-03, 3.19926714e-03, 2.84803587e-03, 2.53536449e-03,
       2.25701972e-03, 2.00923300e-03, 1.78864953e-03, 1.59228279e-03,
       1.41747416e-03, 1.26185688e-03, 1.12332403e-03, 1.00000000e-03])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_squared_error', verbose=0)
```

The best Lasso alpha-value is $\alpha \approx 10^{-1.03}$

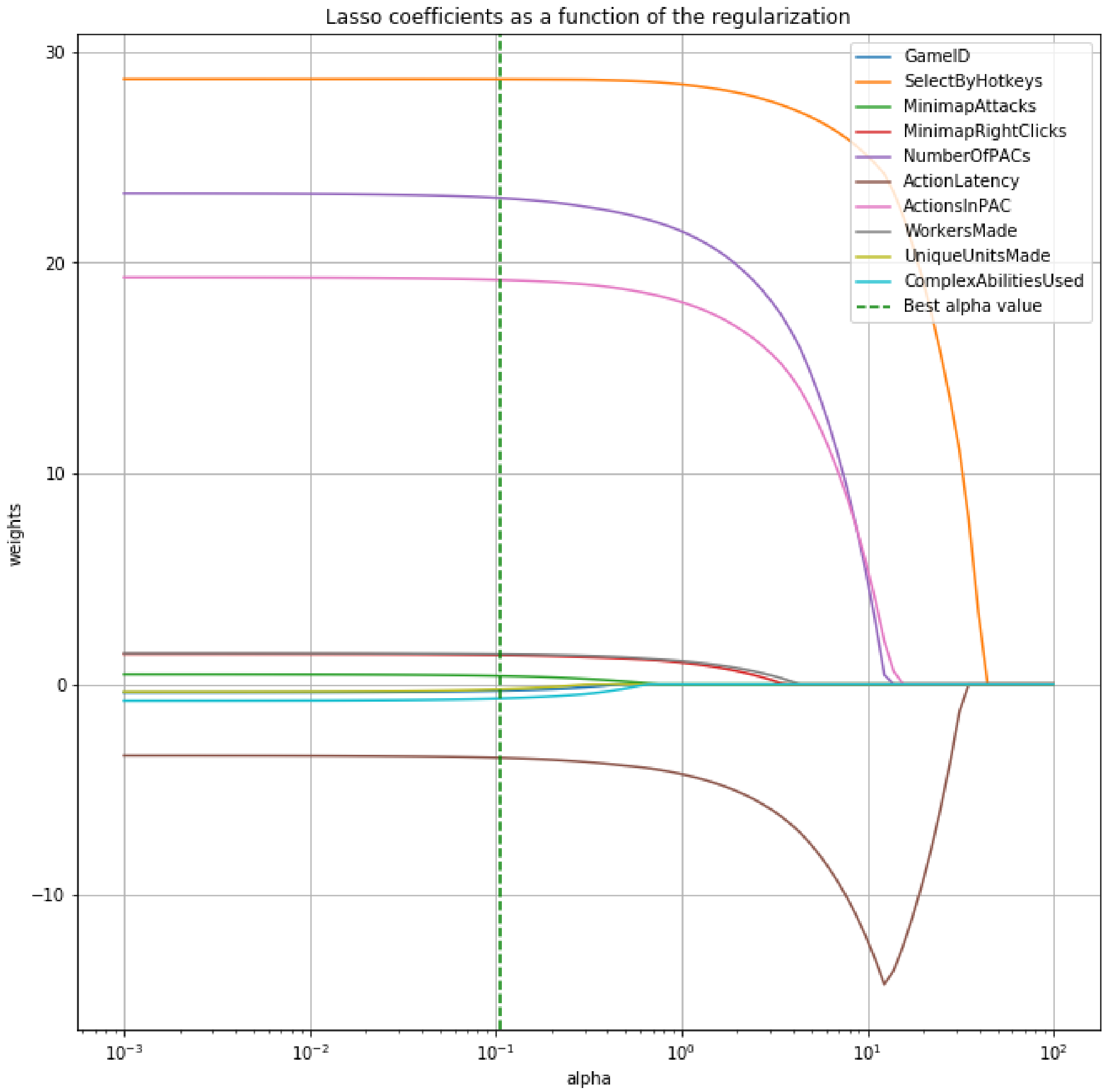
[28]:
```python
-lasso_grid.best_score_
```

[28]:
```
67.6622100008338
```

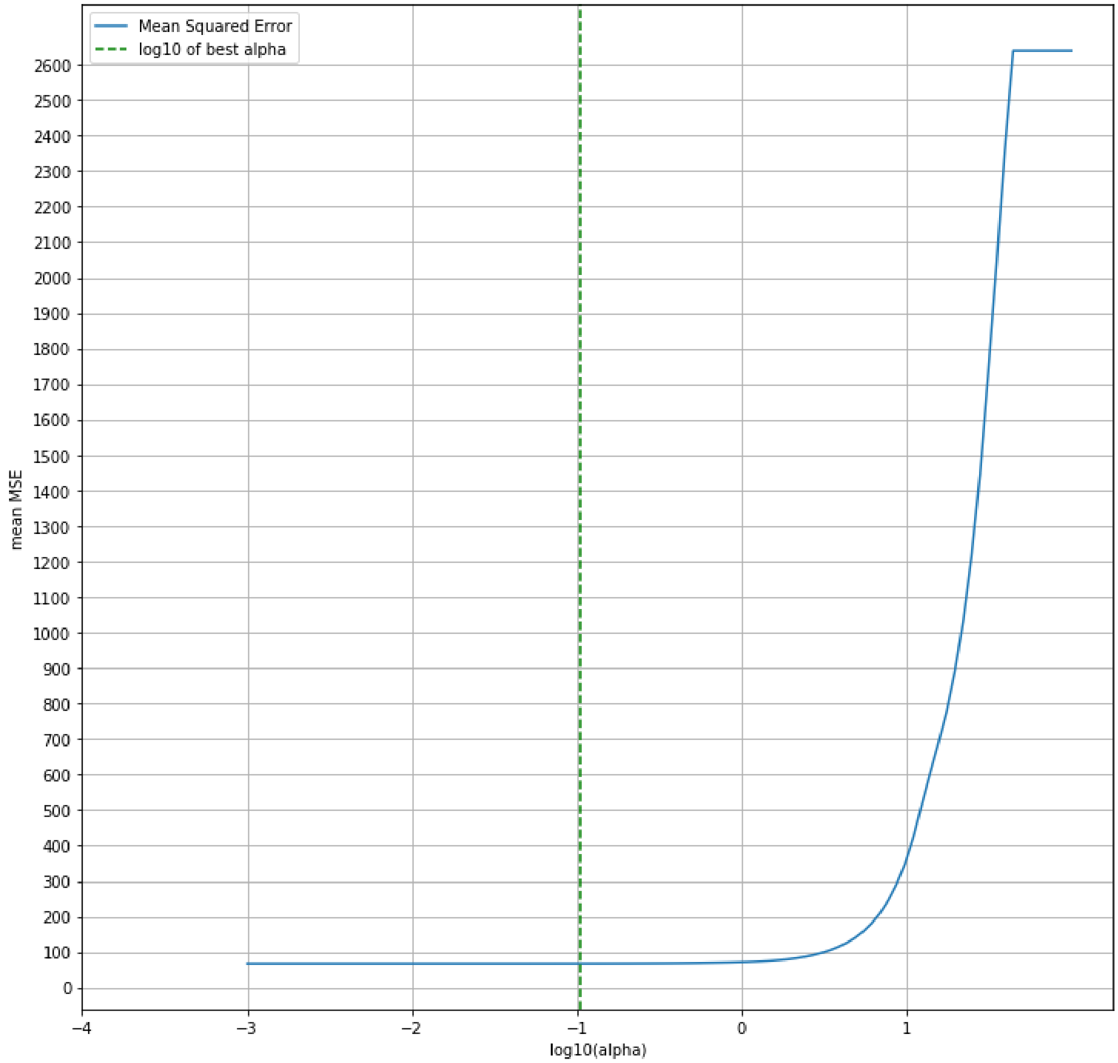
[29]:
```python
lasso_best_alpha = lasso_grid.best_estimator_.alpha
np.log10(lasso_best_alpha)
```

[29]: -0.9797979797979797

[30]:
```
coefs = []
errors = {}
for a in lasso_alphas:
    lasso = Lasso(alpha=a)
    lasso.fit(train_X[predictors], train_y)
    coefs.append(lasso.coef_)
    errors[np.log10(a)] = -np.mean(cross_val_score(lasso, train_X[predictors],␣
 ↪train_y, cv=10,
                               scoring="neg_mean_squared_error"))

plt.figure(figsize=(11,11))

ax = plt.gca()

ax.plot(lasso_alphas, coefs)
ax.set_xscale('log')
#ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Lasso coefficients as a function of the regularization')
plt.axis('tight')

plt.axvline(x=lasso_best_alpha, color='g', linestyle='--')
plt.legend(labels=predictors + ["Best alpha value"])

#plt.yticks(np.arange(-2000, 20000, 1000))
plt.grid()
plt.show()
```




[31]:
```
fig = plt.figure(figsize=(12,12))
ax = plt.plot(list(errors.keys()), list(errors.values()))
plt.grid(True)
plt.yticks(np.arange(0,2700, 100))
plt.xticks(np.arange(-4, 2, 1))
plt.axvline(x=np.log10(lasso_best_alpha), color='g', linestyle='--')
plt.legend(labels=["Mean Squared Error", "log10 of best alpha"])
plt.xlabel("log10(alpha)")
plt.ylabel("mean MSE")
plt.show()
```




**Feature selection with the Lasso**

[32]:
```
lasso = Lasso(alpha=lasso_best_alpha)
lasso.fit(train_X, train_y)
select = SelectFromModel(lasso)
feature_selection = select.fit(train_X, train_y)
selected = feature_selection.transform(train_X)
lasso_selected_columns = train_X.columns[feature_selection.get_support()]
```

[33]:
```
lasso_selected_columns
```

[33]:
```
Index(['GameID', 'Age', 'HoursPerWeek', 'SelectByHotkeys', 'UniqueHotkeys',
       'MinimapAttacks', 'MinimapRightClicks', 'NumberOfPACs', 'ActionLatency',
       'ActionsInPAC', 'WorkersMade', 'UniqueUnitsMade',
       'ComplexAbilitiesUsed'],
      dtype='object')
```

[34]:
```
mse = -np.mean(cross_val_score(lasso, train_X[lasso_selected_columns], train_y,␣
 ↪cv=10, scoring="neg_mean_squared_error"))
print("CV MSE: {}".format(mse))
```

```
CV MSE: 67.59093082352335
```

**Fitting a linear model with the features selected through the Lasso**

[35]:
```
lr = LinearRegression()
mse = -np.mean(cross_val_score(lr, train_X[lasso_selected_columns], train_y,␣
 ↪cv=10, scoring="neg_mean_squared_error"))
print("CV MSE: {}".format(mse))
```

```
CV MSE: 93.11584938032453
```

### 0.2.2 Ridge Regression

[36]:
```
ridge_alphas = 10**np.linspace(-3, 8, 100)

parameters = {'alpha' : ridge_alphas}
ridge_grid = GridSearchCV(Ridge(random_state=42), param_grid=parameters, cv=10,␣
 ↪n_jobs=-1, scoring='neg_mean_squared_error')
ridge_grid.fit(train_X, train_y)
```

```
/home/den/anaconda3/envs/deeplearning/lib/python3.7/site-
packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[36]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=None, normalize=False, random_state=42,
                             solver='auto', tol=0.001),
             iid='warn', n_jobs=-1,
             param_grid={'alpha': array([1.00000000e-03, 1.29154967e-03,
1.66810054e-03, 2.15443469e-03,
       2.78255940e-03, 3.59381366e-03, 4.64158883e-03, 5.99484250e-03,
       7.7...
       7.74263683e+05, 1.00000000e+06, 1.29154967e+06, 1.66810054e+06,
       2.15443469e+06, 2.78255940e+06, 3.59381366e+06, 4.64158883e+06,
       5.99484250e+06, 7.74263683e+06, 1.00000000e+07, 1.29154967e+07,
       1.66810054e+07, 2.15443469e+07, 2.78255940e+07, 3.59381366e+07,
       4.64158883e+07, 5.99484250e+07, 7.74263683e+07, 1.00000000e+08])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_squared_error', verbose=0)
```

Ridge Regression has the lowest cross-validation MSE in our analysis.

[37]:
```
-ridge_grid.best_score_
```

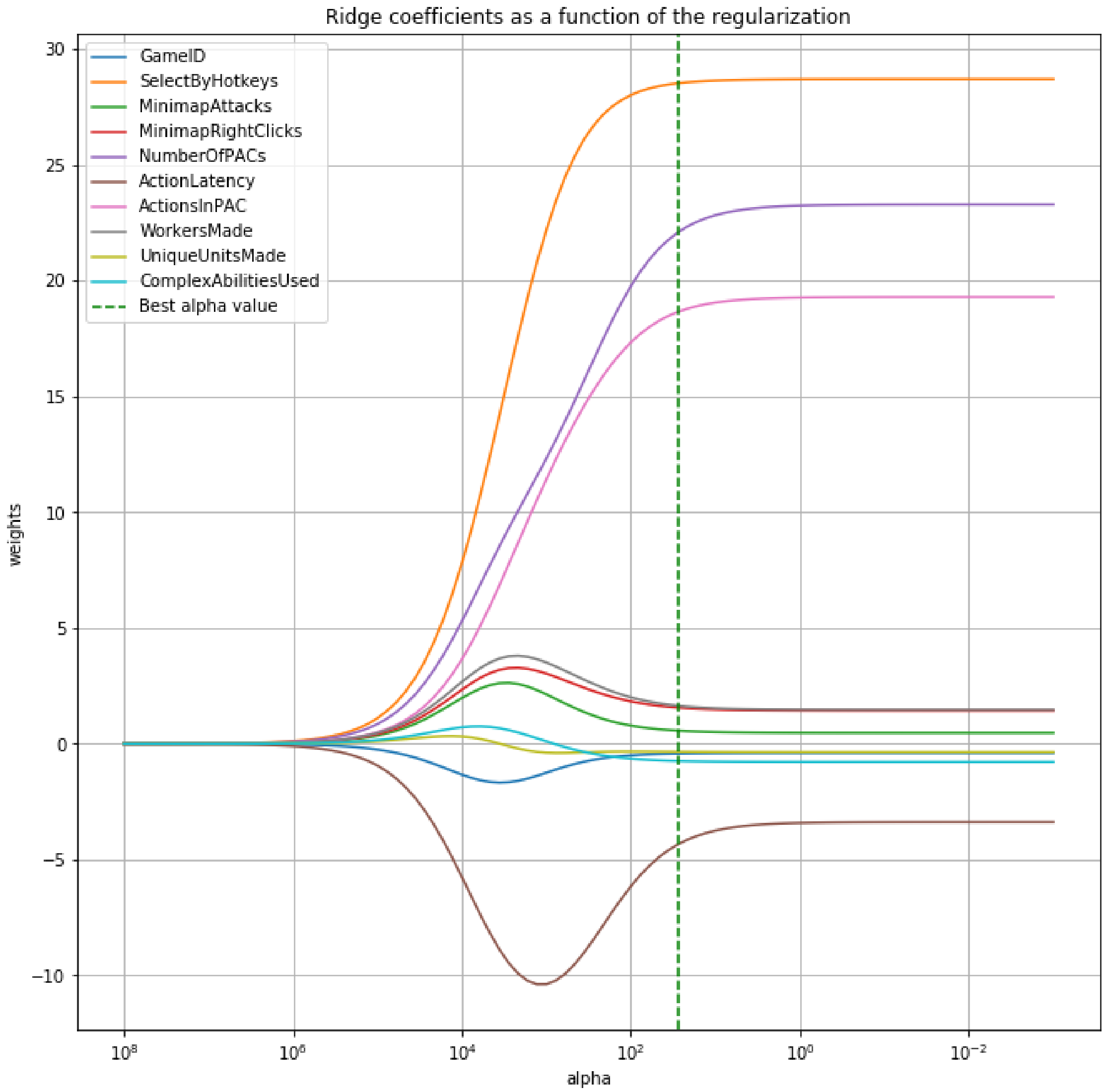
[37]: 68.88458586935693

The best Ridge Regression alpha is $\alpha \approx 27.83$

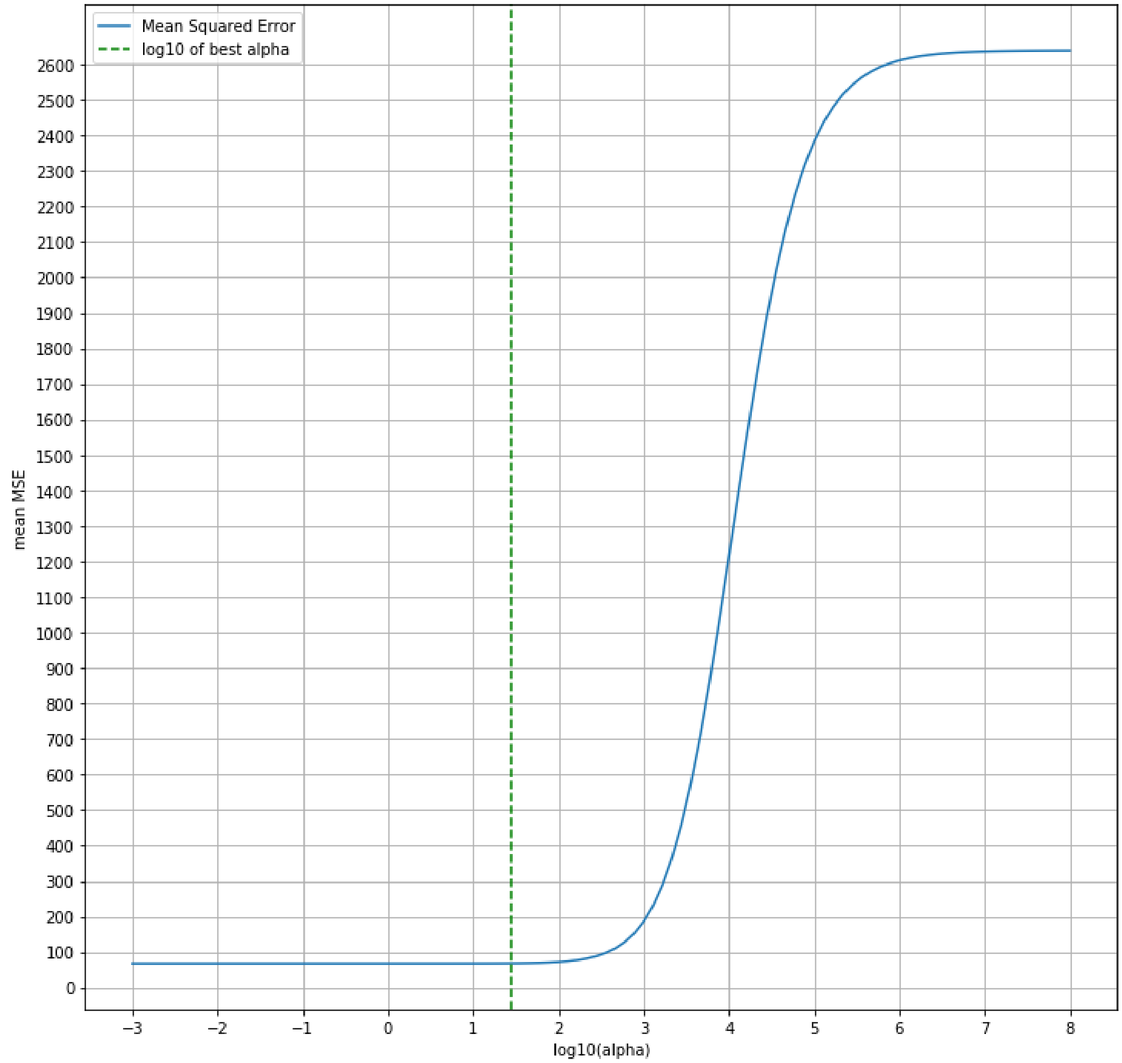
[38]:
```
ridge_best_alpha = ridge_grid.best_estimator_.alpha
ridge_best_alpha
```

[38]: 27.825594022071257

[39]:
```
coefs = []
errors = {}
for a in ridge_alphas:
    ridge = Ridge(alpha=a)
    ridge.fit(train_X[predictors],train_y)
    coefs.append(ridge.coef_)
    errors[np.log10(a)] = -np.mean(cross_val_score(ridge, train_X[predictors],␣
 ↪train_y, cv=10,
                                   scoring="neg_mean_squared_error"))


plt.figure(figsize=(11,11))
ax = plt.gca()
ax.plot(ridge_alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axvline(x=ridge_best_alpha, color='g', linestyle='--')
plt.axis('tight')
plt.grid()
plt.legend(labels=predictors + ["Best alpha value"])
plt.show()
```




[40]:
```
fig = plt.figure(figsize=(12,12))
ax = plt.plot(list(errors.keys()), list(errors.values()))
plt.xlabel("log10(alpha)")
plt.grid()
plt.ylabel("mean MSE")
plt.yticks(np.arange(0,2700, 100))
plt.xticks(np.arange(-3, 9, 1))
plt.axvline(x=np.log10(ridge_best_alpha), color='g', linestyle='--')
plt.legend(labels=["Mean Squared Error"] + ["log10 of best alpha"])
plt.show()
```




**Feature selection with Ridge regression**

[41]:
```
ridge = Ridge(alpha=ridge_best_alpha)
ridge.fit(train_X, train_y)
select = SelectFromModel(ridge)
feature_selection = select.fit(train_X, train_y)
selected = feature_selection.transform(train_X)
ridge_selected_columns = train_X.columns[feature_selection.get_support()]
print("Most important features selected by Ridge Regression:")
print(ridge_selected_columns)
```

```
Most important features selected by Ridge Regression:
Index(['SelectByHotkeys', 'NumberOfPACs', 'ActionsInPAC'], dtype='object')
```

[42]:
```
mse = -np.mean(cross_val_score(ridge, train_X[ridge_selected_columns], train_y,␣
↪cv=10, scoring="neg_mean_squared_error"))
print("CV MSE: {}".format(mse))
```

```
CV MSE: 73.96837409589452
```

**Fitting a polynomial model with the features selected by Ridge Regression**

[43]:
```
errors = {}
for i in range(1,5):
    pr = PolynomialRegression(degree=i)
    mse = -np.mean(cross_val_score(pr, train_X[ridge_selected_columns], train_y,␣
↪cv=10, scoring="neg_mean_squared_error"))
    errors[i] = mse
pd.Series(errors).sort_values()
```

[43]:
```
2    11.668768
3    11.721620
4    11.862008
1    73.782559
dtype: float64
```

# 1 Predicting on the test set

We have now found the best predictor for univariate linear regression vs. the response APM and optimal values for alpha in terms of Lasso and Ridge Regression. Let's see how models with those parameters perform on the test set.

[44]:
```
test_X = pd.DataFrame(preprocessor.fit_transform(test_X), columns=test_X.columns)
```

## 1.1 Univariate testing

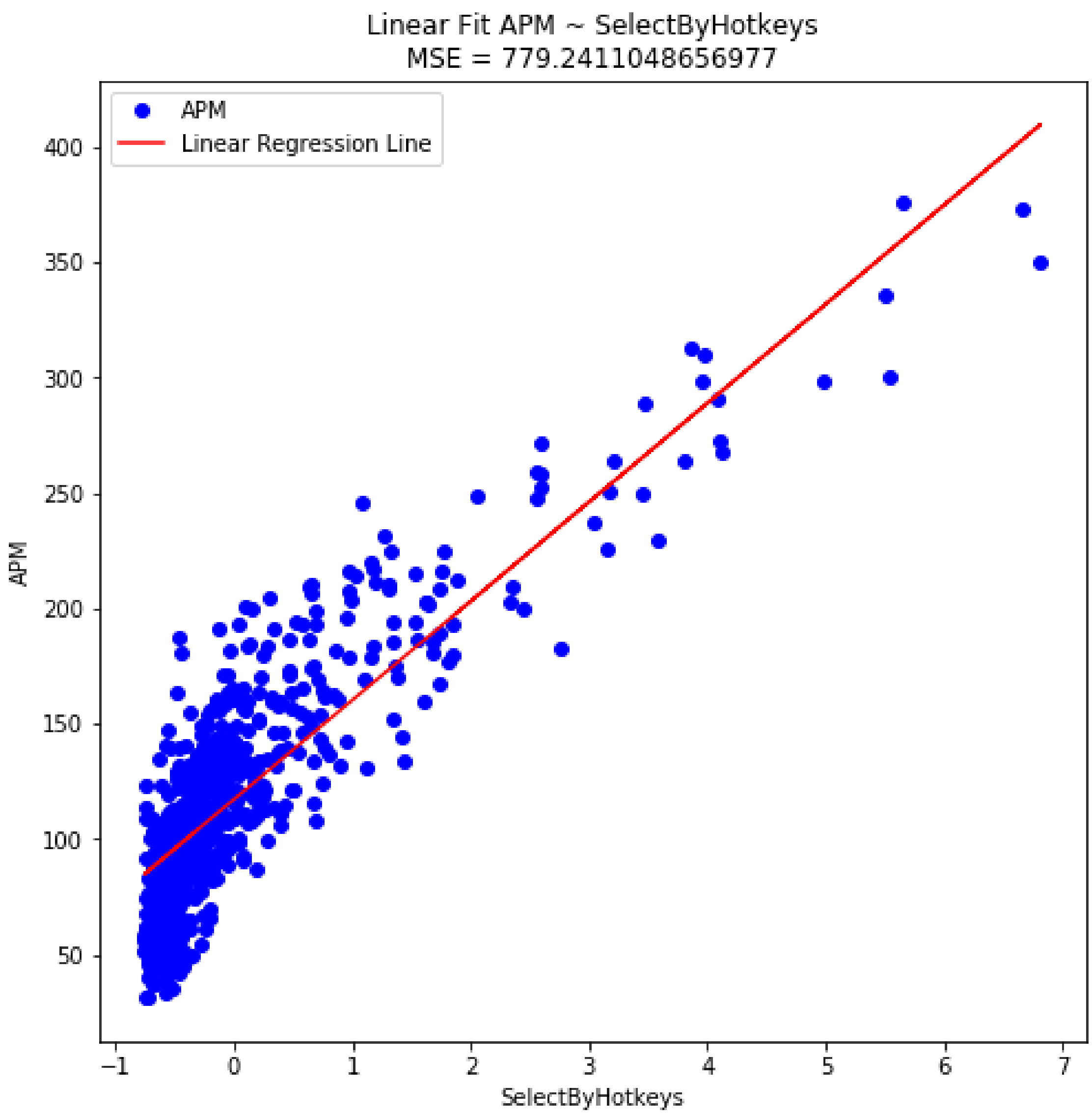
### 1.1.1 Linear Regression

[45]:
```
lr = LinearRegression()
lr.fit(train_X[["SelectByHotkeys"]], train_y)
pred = lr.predict(test_X[["SelectByHotkeys"]])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}\nr-squared: {}".format(mse, r2))
```

```
MSE: 779.2411048656977
r-squared: 0.7343884756211243
```

[46]:
```
fig = plt.figure(figsize=(8,8))

plt.title("Linear Fit APM ~ SelectByHotkeys\nMSE = {}".format(mse))
```

```
plt.xlabel("SelectByHotkeys")
plt.ylabel("APM")
plt.plot(test_X["SelectByHotkeys"], test_y, 'bo', label="APM")
plt.plot(test_X["SelectByHotkeys"], pred, 'r-', label="Linear Regression Line")
plt.legend()
plt.show()
```



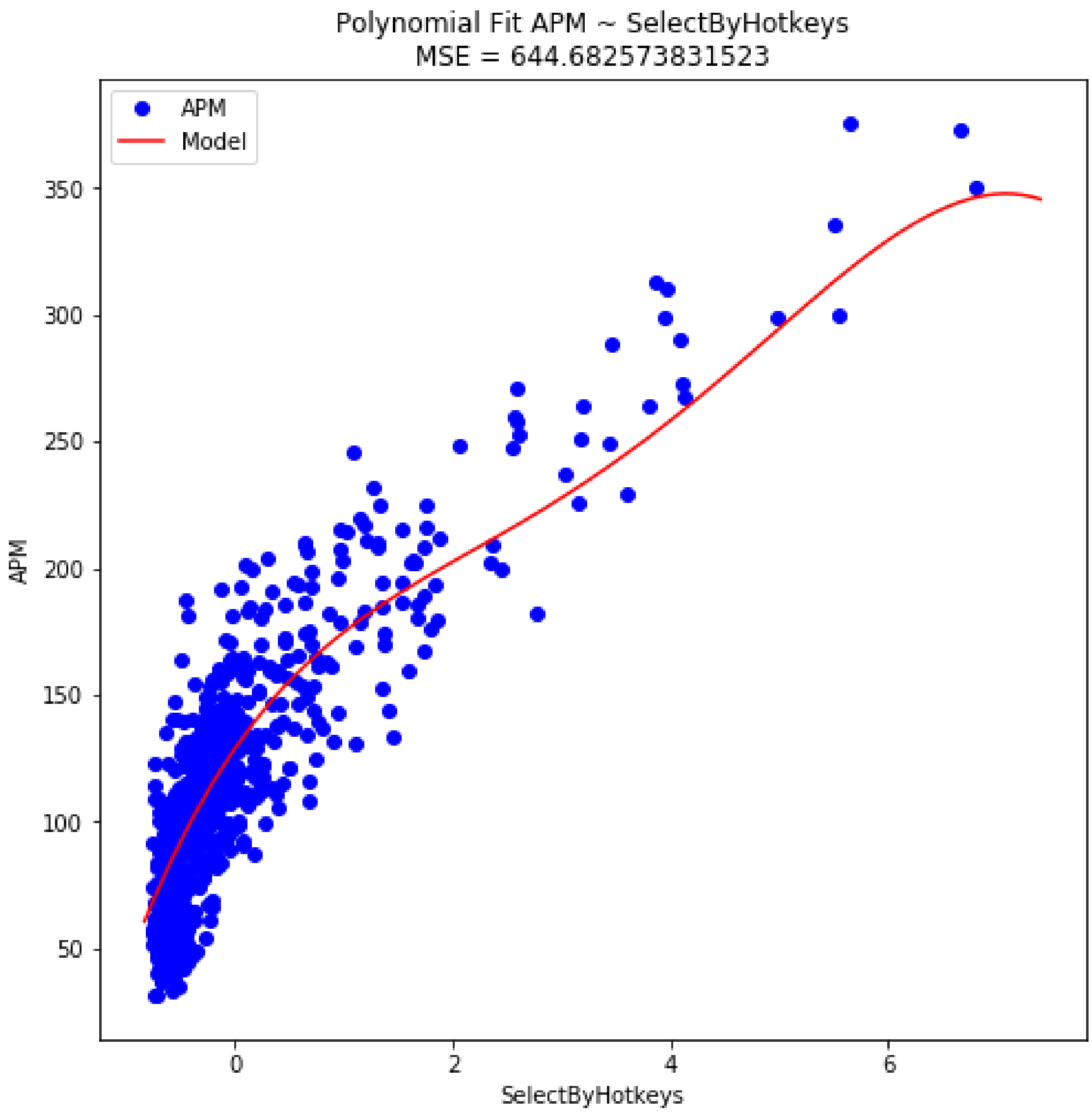

### 1.1.2 Polynomial Regression

We found that `SelectByHotkeys` had the lowest training MSE in terms of univariate polynomial regression with degree 4.

[47]:
```
pr = PolynomialRegression(4)

pr.fit(train_X[["SelectByHotkeys"]], train_y)
pred = pr.predict(test_X[["SelectByHotkeys"]])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}\nr-squared: {}".format(mse, r2))
```

```
MSE: 644.682573831523
r-squared: 0.7802539931393888
```

[48]:
```
fig = plt.figure(figsize=(8,8))

plt.title("Polynomial Fit APM ~ SelectByHotkeys\nMSE = {}".format(mse))
plt.xlabel("SelectByHotkeys")
plt.ylabel("APM")
plt.plot(test_X["SelectByHotkeys"], test_y, 'bo', label="APM")
xs = np.linspace(np.min(train_X[predictor]), np.max(train_X[predictor]), 100)
plt.plot(xs, pr.predict(xs.reshape(-1,1)), 'r-', label="Model")
plt.legend()
plt.show()
```




## 1.2 Multivariate testing

### 1.2.1 The Lasso

[49]:
```
lasso = Lasso(alpha=lasso_best_alpha, random_state=42)
lasso.fit(train_X, train_y)
pred = lasso.predict(test_X)
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 67.80077211461267
r-squared: 0.9768894809026651
```

**Testing with a linear model using the features we selected with the Lasso**

[50]:
```
lr = LinearRegression()
lr.fit(train_X[lasso_selected_columns], train_y)
pred = lr.predict(test_X[lasso_selected_columns])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 67.07687233281382
r-squared: 0.9771362288261776
```

**Testing with a linear model using the features we selected by backward selection**

[51]:
```
lr.fit(train_X[predictors], train_y)
pred = lr.predict(test_X[predictors])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 67.20822745303786
r-squared: 0.9770914552208071
```

### 1.2.2 Ridge Regression

With Ridge Regression we also could reduce the training MSE by a lot. Let's finally check how our Ridge Regression model performs on test data.

[52]:
```
ridge = Ridge(alpha=ridge_best_alpha)
ridge.fit(train_X, train_y)
pred = ridge.predict(test_X)
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}\nr-squared: {}".format(mse, r2))
```

```
MSE: 69.36678817493554
r-squared: 0.9763556898713824
```

**Using the features we selected by Ridge Regression**

[53]:
```
ridge.fit(train_X[ridge_selected_columns], train_y)
pred = ridge.predict(test_X[ridge_selected_columns])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
```

```
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 74.7260309360541
r-squared: 0.9745289425008846
```

**Testing with a polynomial model using the features we selected through Ridge Regression**

[54]:
```
pr = PolynomialRegression(degree=2)
pr.fit(train_X[ridge_selected_columns], train_y)
pred = pr.predict(test_X[ridge_selected_columns])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 18.492197371313157
r-squared: 0.9936967638100199
```

**Testing with a polynomial model using the features we selected by backward selection**

[55]:
```
pr.fit(train_X[predictors], train_y)
pred = pr.predict(test_X[predictors])
mse = mean_squared_error(test_y, pred)
r2 = r2_score(test_y, pred)
print("MSE: {}".format(mse))
print("r-squared: {}".format(r2))
```

```
MSE: 16.29545641916544
r-squared: 0.994445543243397
```

## 2 Conclusion

The best model we found to predict `APM` for a given observation is a polynomial regression model of degree 2 utilizing the features we found through backward selection. Here we found a test MSE of 16.30 and a $r^2$ value of 99.44%. This means the model explains 99.44% of the response variance.