Predicting NHL prospects' first season points with different regression methods

1 Introduction

When a new player is drafted into the NHL, there are a lot of predictions about how the rookies will perform. The draft number is one of the most essential factors for the expectation of points. A draft number represents how talented and potential a prospect is based on NHL teams' opinion. The number one draft is predicted to be the most talented of the draft year. In this project, I am trying to predict how the draft number affects players' points during the next season. In Section 2, I will further clarify the problem at hand. After that, in section 3, I will focus more on the data and machine learning methods. The results of those methods are discussed in Section 4, and after that, in Section 5, I will wrap up final thoughts and conclusions on this problem.

2 Problem Formulation

I am trying to predict how the draft number of an NHL player will affect the points for the next season after the draft. The data points for this problem are NHL players that are drafted. The datapoints are from the NHL API [1], and the information is based on a suitable combination of draft, prospect, and people data. Draft data is used to get the draft year and all the prospects drafted during that year. NHL players' stats can be fetched from people data and prospect data is used to link draft data and people data. As the API provides comprehensive data, it can be simplified to have only the draft number and points from the next season. The label for this problem is the prospect's regular season points. The feature of the datapoint is the draft number. Both the feature and the label are integers.

2.1 Summary of the Problem

Datapoint: NHL player that has been drafted, Label: season points for the player, Feature: draft number.

3 Methods

3.1 Data collection and preprocessing

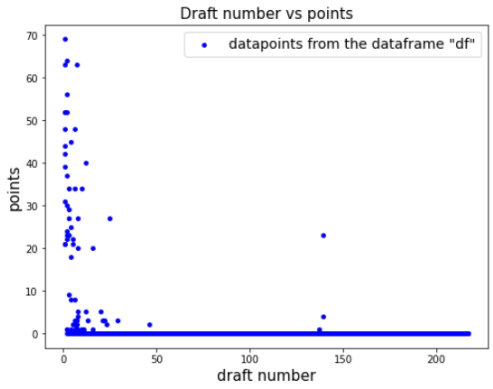
I decided to get the prospect data for the years 2010–2020 to get quite modern data as the game in the NHL is way different now than in the 1990's. 11 seasons with 2343 drafted players (datapoints) should give enough data to make accurate conclusions. To collect and preprocess the data to be used for this project, I needed to make a couple thousand GET requests to the NHL API [1]. Due to the massive number of GET requests, I decided to use parallel computing to decrease the preprocessing time. From draft data, I was able to get the draft number for the player and, from the player's season stats, the points for the following season. The linking of people and draft data was done with API's prospect data. After linking those, the data was ready for ML methods. There were around 10 players that were missing some of the data, so I was able to check from elite prospects [2] (a hockey stats site) that they had 0 points the following season, so I added those also.

3.2 Feature selection

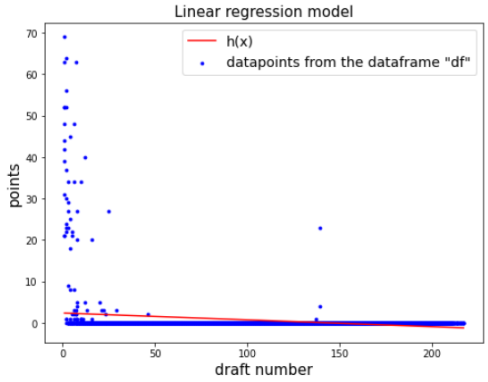
When it comes to NHL players' first season, the draft position is usually the biggest indicator of the next season's success, as NHL teams are drafting the most talented players as early as possible. There are also a lot of predictions from sports reporters before seasons trying to predict the points for top rookies, so I wanted to see how ML methods compete with those. I also wanted to do two-dimensional plots to visualize the data, so one numeric feature with one numeric label works well with it.

3.3 Machine learning models

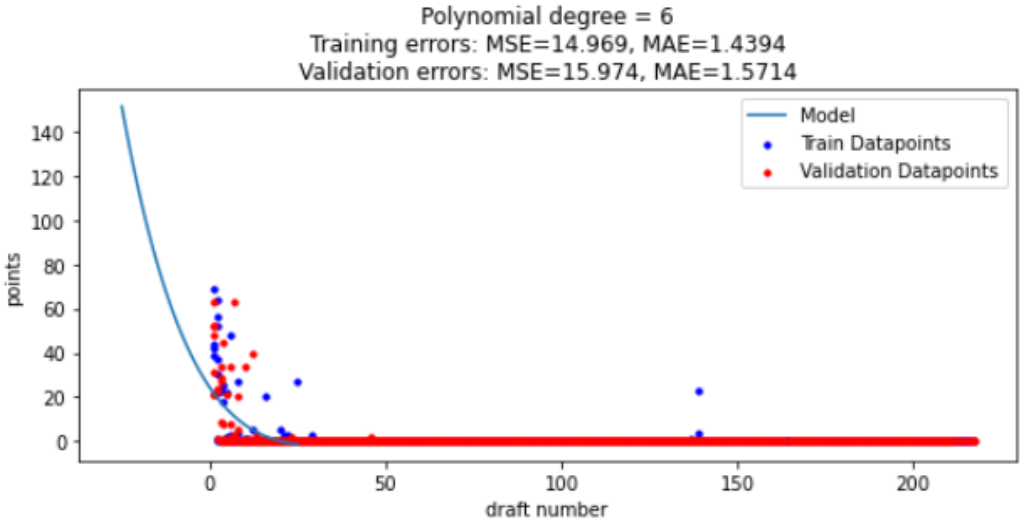
As I needed to get the number of points as a label value, it was clear that I needed to use a model that supported that. Therefore, I was looking into linear methods instead of classification methods. At first, I wanted to draw the datapoints to get a better picture of the data (picture 3.1). I first tested with linear regression as I was expecting that there would be a correlation between points and draft number as early drafted players usually perform better. After plotting, it was quickly apparent that linear regression would not be a great fit for it as the 0-point data was affecting it too much. So next, I decided to use polynomial regression as it can handle 0-point data better. With polynomial regression, I was able to get the plot to fit the points. I did a lot of testing with the degrees and found that degree 6 was the best for this data (picture 3.3). That also minimized training and validation errors. Polynomial regression had a way smaller training error compared to linear regression (15<23), so I decided to move forward with polynomial regression to loss function selection.



picture 3.1



picture 3.2



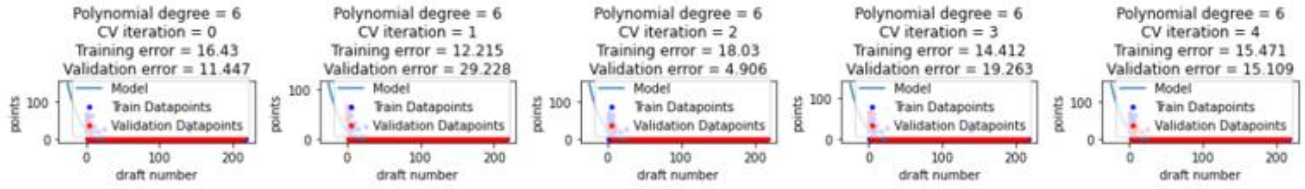
picture 3.3

3.4 Loss functions

After seeing that polynomial regression was more suitable for this problem, I chose two candidates for a loss function, mean squared error (MSE) and mean absolute error (MAE). Both loss functions work well with polynomial regression. I also found those easier to understand than r2 score, as they are always non-negative. Mean squared error was also familiar as we have used it during the course exercises and it seems to be one of most popular loss functions[3] as well as the mean absolute error. They were also included in the Sklearn library. The loss functions were used to calculate the training and validation errors. Using the training and validation errors, I was able to choose which polynomial degree was best for this data. When choosing which loss function to choose, the main question was how much I would penalize large errors, as MSE will be affected more by those due to squering values. For this problem, large errors are quite undesirable, so I decided to continue with mean squared error as it favored a plot that is closer to actual data. MAE did favor linear regression as it was affected by 0-point data too much and therefore was not the best call for this project.

3.5 Model validation

I chose K-fold cross validation for the maximum accuracy. Instead of single split data, K-fold uses all the data for both test and training data instead of one or the other, so it is more accurate. Even though K-fold is heavier and slower than single split validation, it was not too slow to make a difference in favor of single split. Before using K-fold cross validation, I took 20% of the data out for the test set and used the rest 80% for K-fold. I used k = 5 as it allowed a good combination of plot visibility and accuracy. By selecting k = 5, the training set will contain 80% of the data points that are not in the test set. The validation set will contain the remaining 20% left over from the training set.




```
Degree 6, avg train error = 15.31130, avg val error = 15.99061, test error = 15.33866
```

picture 3.3

4 Results

In the previous section, I chose to use mean squared error as the loss function and K-fold cross validation to get accurate model validation. My ML model of choice was polynomial regression, but to justify the choice, I decided to include a degree of 1 to compare it to linear regression as well. Degrees 1 to 7 were considered in this project as higher degrees were not giving any extra value for this problem. I also included test errors for all the data. The test sets were constructed by using a single split and consisted of randomly shuffled 20% of the data. With the remaining 80% I was able to use K-fold to get the training and validation errors. So, the test set has not been used to train the ML methods.

After all the data and ML method preparation, we are ready for the results. Looking at the picture (4.1), it is clear to see that degree 6 is the most optimal degree for polynomial regression. It beats the other degrees in training, validation, and test errors. The plot with degree 6 was also the closest to the data points.

```
Degree 1, avg train error = 23.05769, avg val error = 23.22407, test error = 24.06858
Degree 2, avg train error = 21.65520, avg val error = 21.98704, test error = 22.19120
Degree 3, avg train error = 19.93868, avg val error = 20.44851, test error = 20.14778
Degree 4, avg train error = 18.23494, avg val error = 18.81470, test error = 18.20946
Degree 5, avg train error = 16.69276, avg val error = 17.33745, test error = 16.62991
Degree 6, avg train error = 15.31130, avg val error = 15.99061, test error = 15.33866
Degree 7, avg train error = 23.93214, avg val error = 23.95784, test error = 24.93285
```

picture 4.1

5 Conclusions

Overall, the ML models, loss functions, and validations were quite optimal for this project. With K-fold and a couple thousand data points, I was able to get accurate model validation and all the errors of selected degree 6 were close to each other, so there was no massive overfitting or underfitting. One main reason for that was the vast number of data points. Also, the technical side of data preprocessing was successful, and I got to train parallel computing with GET requests. On the other hand, the main difficulty I identified for the problem at hand was that only the top prospects would receive any points at all, with the majority of prospects receiving 0 points. Therefore, it is challenging to find a well-matched model. Also, the data varies a lot from season to season and player to player, so an exact matching model cannot be found. Under these conditions, the model is performing quite well. However, for the future, as most of the players drafted with a number over 30 will not get any points for the first season, one idea for the future could be to split the data for the top players and then for the rest and use different models for them.

6 References

https://github.com/dword4/nhlapi [1]

https://www.eliteprospects.com/ [2]

https://analyticsindiamag.com/most-used-loss-functions-to-optimize-machine-learning-algorithms/ [3]

CS-C3240 Machine Learning D, Course material and code assignments [4]

7 Appendices

project

March 28, 2022

[1]:
```
# Hi, welcome to my ML project related to NHL rookie year points based on the␣
 ↪draft number.
# At the end you can see K-Fold example with the best model. Prior that there␣
 ↪are also linear
# regression tested and polynomial with different validation methods.
```

[2]:
```
import pandas as pd    # library for data manipulation and analysis
import multiprocessing as mp # using parallel computing to reduce time as I do␣
 ↪multiple get requests
import requests # for get requests
import matplotlib.pyplot as plt    # library providing tools for plotting data
import numpy as np    # library for numerical computations (vectors, matrices,␣
 ↪tensors)

from sklearn.preprocessing import PolynomialFeatures    # function to generate␣
 ↪polynomial and interaction features
from sklearn.linear_model import LinearRegression, HuberRegressor    # classes␣
 ↪providing Linear Regression with ordinary squared error loss and Huber loss,␣
 ↪respectively
from sklearn.metrics import mean_squared_error,mean_absolute_error    #␣
 ↪function to calculate mean squared error

from functools import reduce # helper function for code readability
from sklearn.model_selection import train_test_split, KFold
```

[3]:
```
base_url = "https://statsapi.web.nhl.com/"
years = range(2010,2021)
def draftPlayersByYear(year):
    prospect_data = requests.get(f"{base_url}/api/v1/draft/{year}").json()
    nested_list = list(map(lambda round:␣
 ↪round["picks"],prospect_data["drafts"][0]["rounds"]))
    players = reduce(lambda a,b:a+b, nested_list)
    return players
#pool is used to use all the cores of the cpu
pool = mp.Pool(processes=mp.cpu_count())
players = pool.map(draftPlayersByYear,years)
players = reduce(lambda a,b:a+b, players)
```

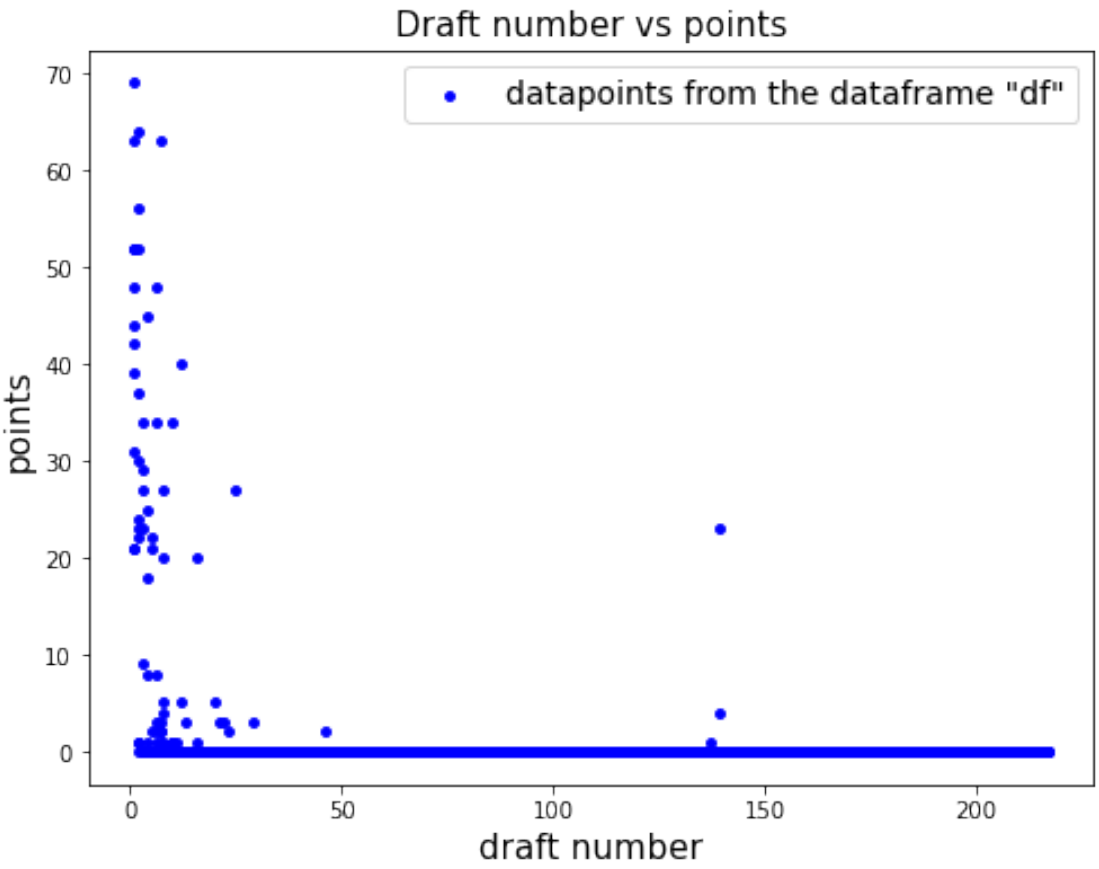
```
#number of data points.
print(len(players))
pool.close()
pool.join()
```

2343

[4]:
```
# importing the requests library


# par
def filter_player_data(player):
    if not player["prospect"].get("id"):
        data_point = {"points":0,"draft_year":player["year"], "draft_number":␣
↪player["pickOverall"]}
        return data_point

    player_year=player["year"]
    player_link = player["prospect"]["link"]
    prospectData = requests.get(f"{base_url}{player_link}").json()
    prospect= prospectData.get("prospects")
    if not prospect:
        data_point = {"points":0, "draft_number": player["pickOverall"]}
        return data_point
    if len(prospect)==0:
        data_point = {"points":0, "draft_number": player["pickOverall"]}
        return  data_point
    nhl_player_id = prospect[0].get("nhlPlayerId")
    if not nhl_player_id:
        data_point = {"points":0, "draft_number": player["pickOverall"]}
        return data_point
    player_stats = requests.get(f"{base_url}/api/v1/people/{nhl_player_id}/
↪stats?stats=statsSingleSeason&season={player_year}{player_year+1}").
↪json()["stats"][0]
    player_season_points = 0
    if len(player_stats["splits"])>0:
        player_season_points= player_stats["splits"][0].get("stat",{"points":
↪0}).get("points",0)

    data_point = {"points":player_season_points, "draft_number":␣
↪player["pickOverall"]}

    return data_point

#parallel pool
pool = mp.Pool(processes=mp.cpu_count())
filtered_data = pool.map(filter_player_data,players)
```

```
pool.close()
pool.join()
```

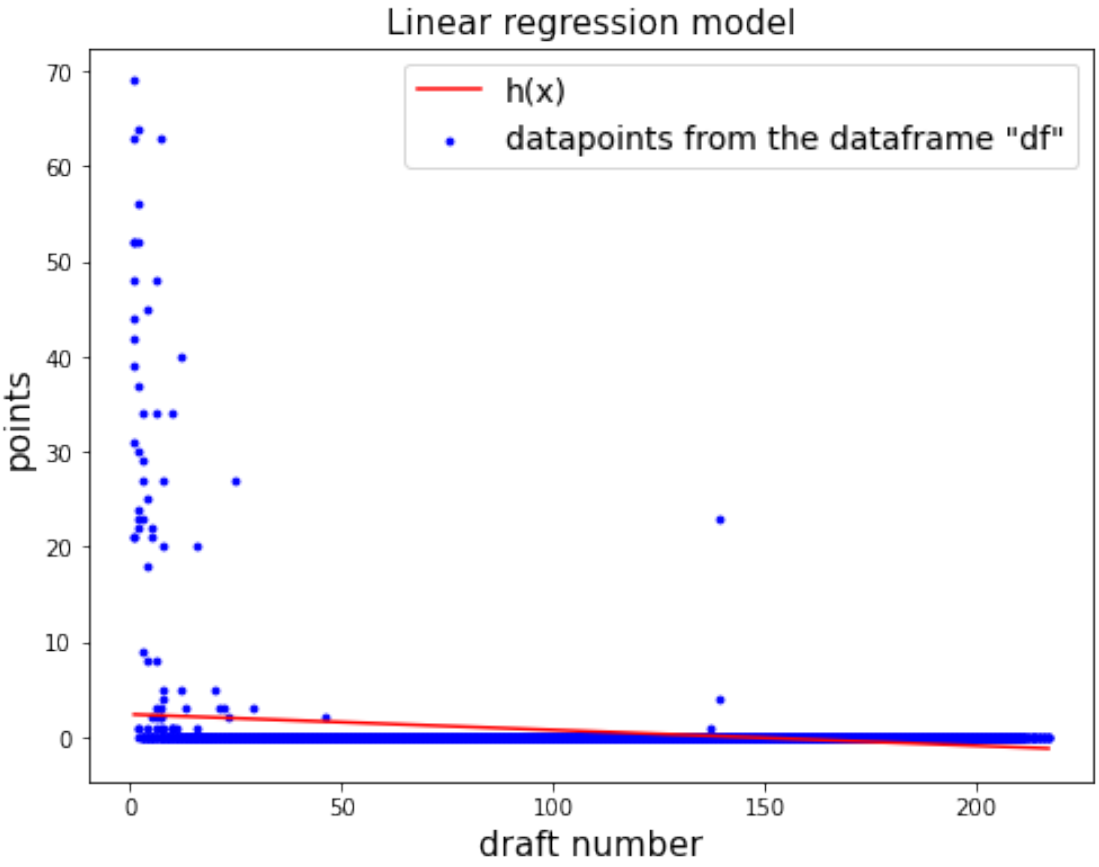
[5]:
```
df = pd.DataFrame.from_dict(filtered_data)
X = df["draft_number"].to_numpy().reshape(-1, 1)
y = df["points"].to_numpy()

plt.figure(figsize=(8, 6))    # create a new figure with size 8*6

# create a scatter plot of datapoints
# each datapoint is depicted by a dot in color 'blue' and size '10'
plt.scatter(X, y, color='b', s=15, label='datapoints from the dataframe "df"')

plt.xlabel('draft number',size=15) # define label for the horizontal axis
plt.ylabel('points',size=15) # define label for the vertical axis

plt.title('Draft number vs points',size=15) # define the title of the plot
plt.legend(loc='best',fontsize=14) # define the location of the legend

plt.show()  # display the plot on the screen
```



[6]:
```
## Fit a linear regression model
regr = LinearRegression()
regr.fit(X, y)

## Predict label values based on features and calculate the training error
y_pred =   regr.predict(X)
tr_error = mean_squared_error(y, y_pred)

print('The training error is: ', tr_error)    # print the training error
```

```
The training error is:  23.26420511206483
```

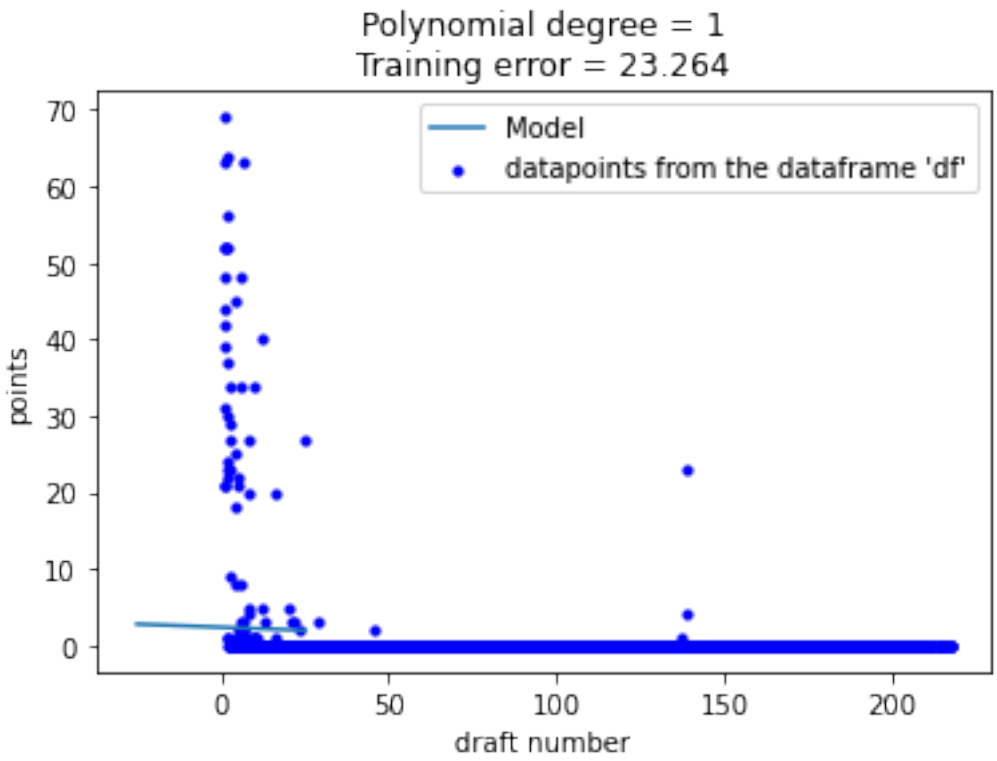
[7]:
```
## visualizing linear regression model

plt.figure(figsize=(8, 6))    # create a new figure with size 8*6

# create a scatter plot of datapoints
# each datapoint is depicted by a dot in color 'blue' and size '10'
plt.scatter(X, y, color='b', s=8, label='datapoints from the dataframe "df"')

# plot the predictions obtained by the learnt linear hypothesis using color␣
 ↪'red' and label the curve as "h(x)"
y_pred = regr.predict(X)    # predict using the linear model
plt.plot(X, y_pred, color='r', label='h(x)')

plt.xlabel('draft number',size=15) # define label for the horizontal axis
plt.ylabel('points',size=15) # define label for the vertical axis

plt.title('Linear regression model',size=15) # define the title of the plot
plt.legend(loc='best',fontsize=14) # define the location of the legend

plt.show()  # display the plot on the screen
```



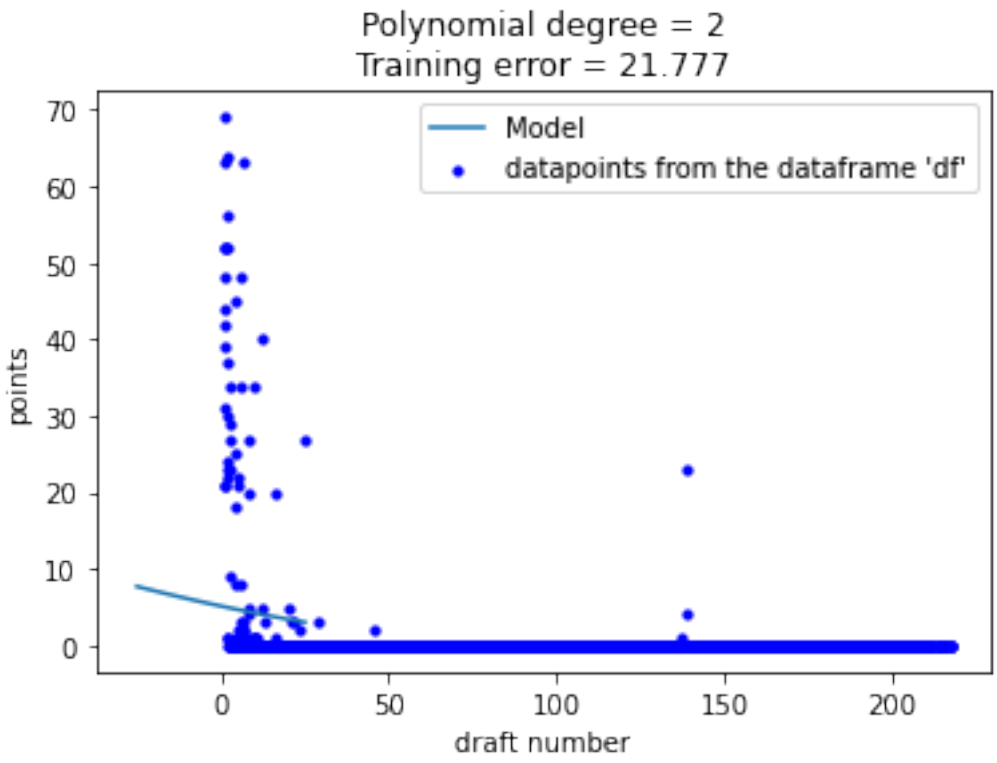
[8]:
```
## define a list of values for polynomial degrees
degrees = [1,2,3,4,5,6,7,8]

# declare a variable to store the resulting training errors for each polynomial␣
 ↪degree
tr_errors = []

for i in range(len(degrees)):     # use for-loop to fit polynomial regression␣
 ↪models with different degrees

    print("Polynomial degree = ",degrees[i])

    poly = PolynomialFeatures(degree=degrees[i])     # initialize a polynomial␣
 ↪feature transformer
    X_poly =poly.fit_transform(X)    # fit and transform the raw features

    lin_regr = LinearRegression(fit_intercept=False) # NOTE:␣
 ↪"fit_intercept=False" as we already have a constant iterm in the new feature␣
 ↪X_poly
```

```
   lin_regr.fit(X_poly,y)    # fit linear regression to these new features and␣
↪labels (labels remain same)

   y_pred = lin_regr.predict(X_poly)    # predict using the learnt linear model
   tr_error = mean_squared_error(y, y_pred)   # calculate the training error
   tr_errors.append(tr_error)
   X_fit = np.linspace(-25, 25, 100)    # generate samples
   plt.plot(X_fit, lin_regr.predict(poly.transform(X_fit.reshape(-1, 1))),␣
↪label="Model")    # plot the polynomial regression model
   plt.scatter(X, y, color="b", s=10, label="datapoints from the dataframe␣
↪'df'")    # plot a scatter plot of y(points) vs. X(draft number) with color␣
↪'blue' and size '10'
   plt.xlabel('draft number')    # set the label for the x/y-axis
   plt.ylabel('points')
   plt.legend(loc="best")    # set the location of the legend
   plt.title('Polynomial degree = {}\nTraining error = {:.5}'.
↪format(degrees[i], tr_error))    # set the title
   plt.show()    # show the plot
```

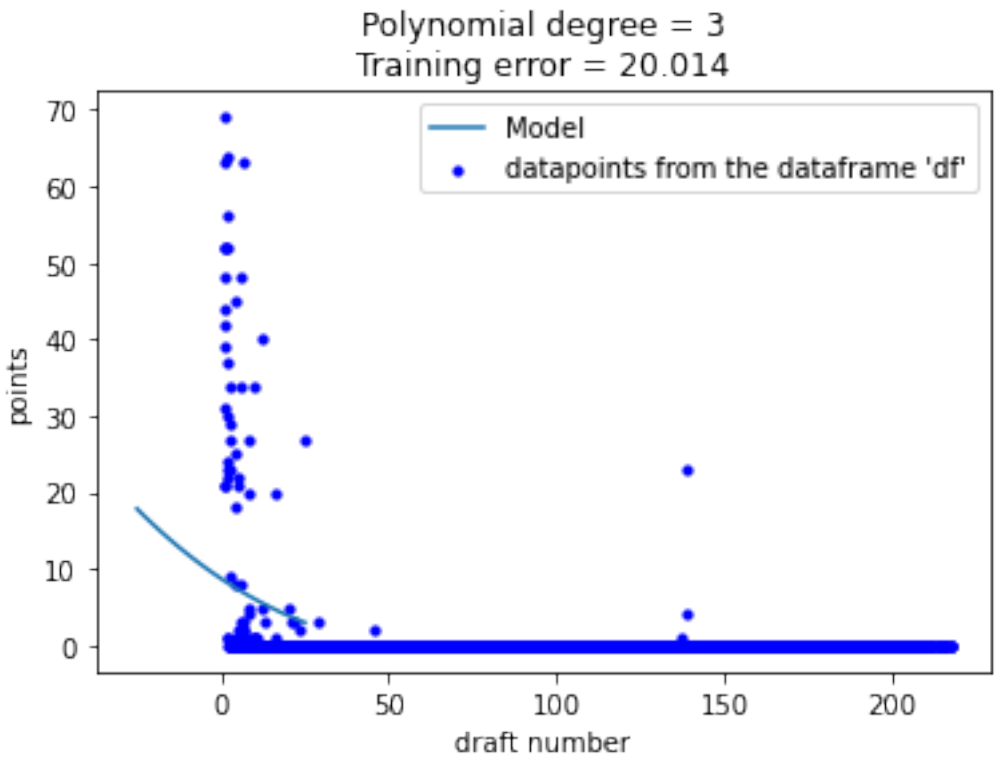
```
Polynomial degree =  1
```



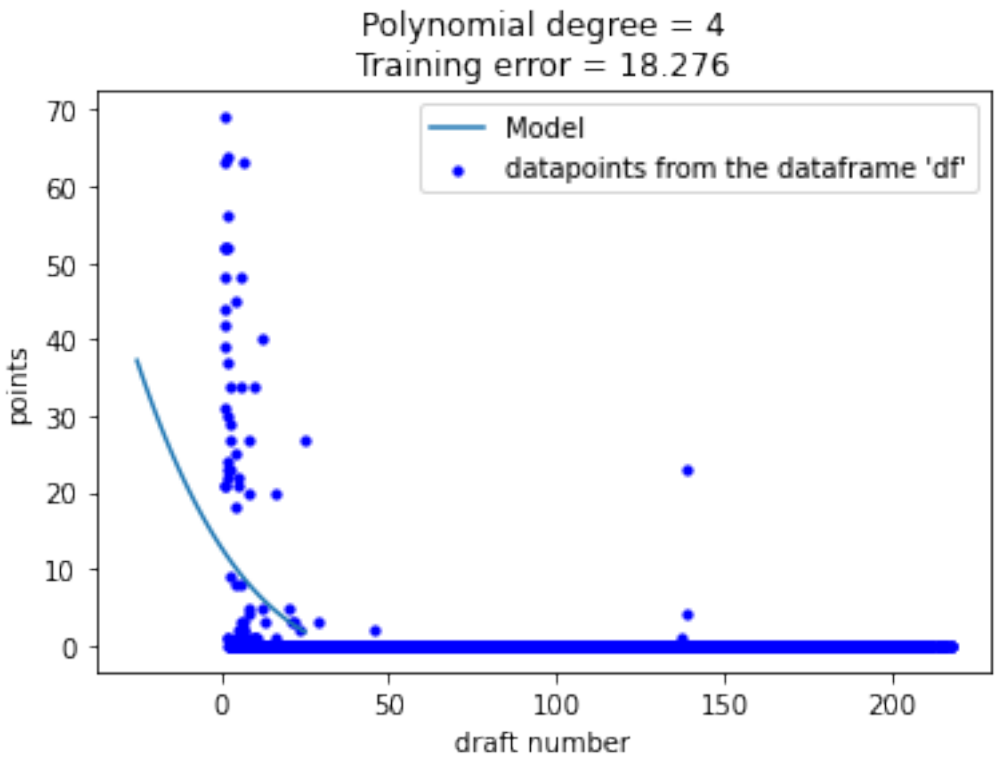

```
Polynomial degree =  2
```



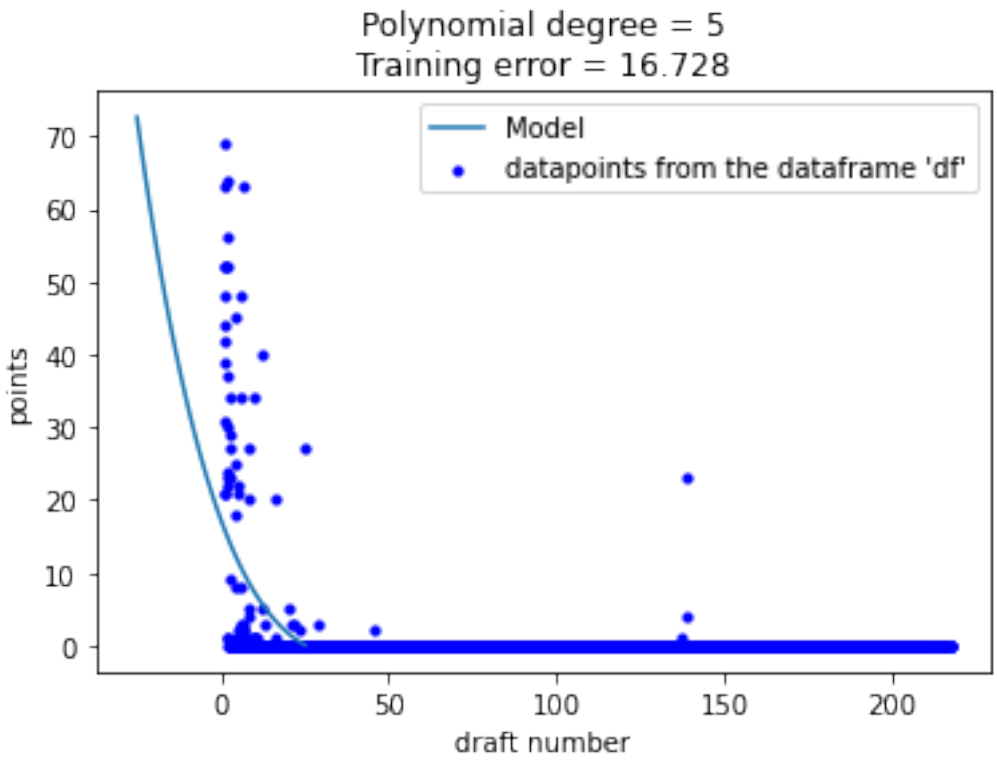

```
Polynomial degree =  3
```



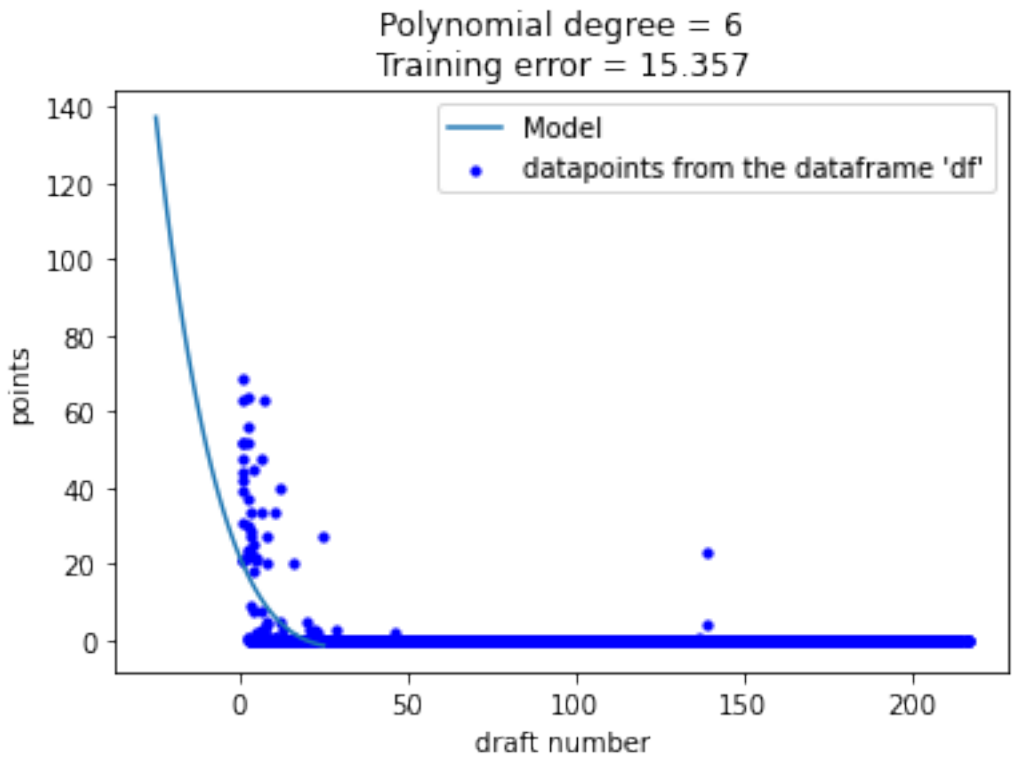

```
Polynomial degree =  4
```



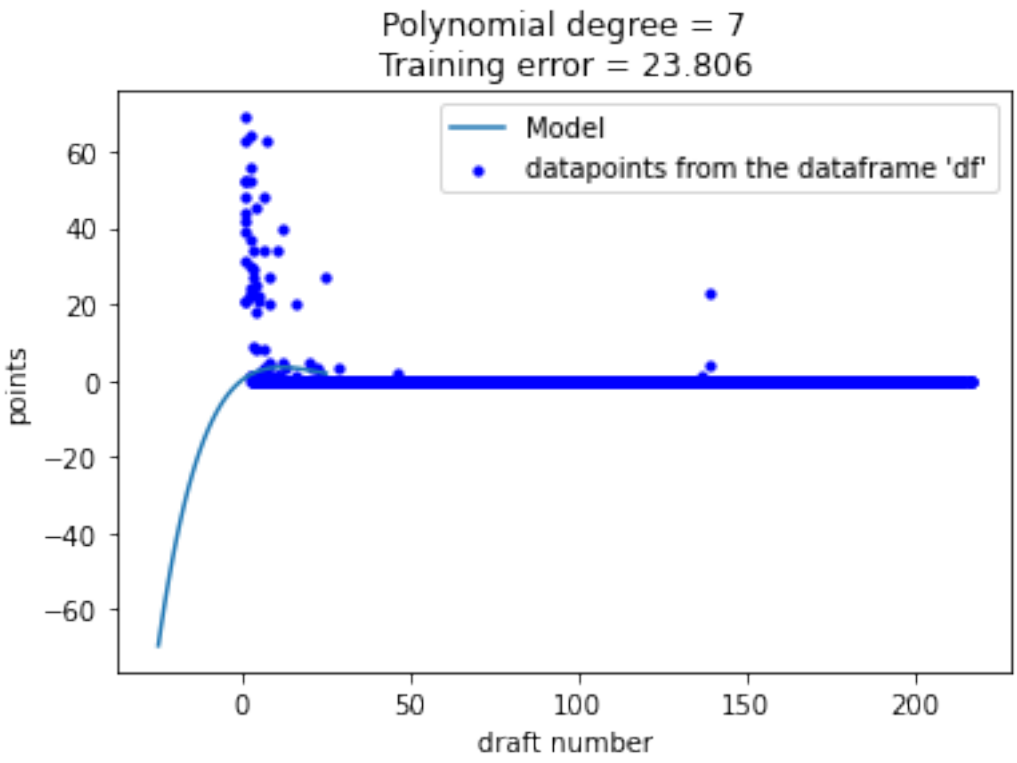

```
Polynomial degree =  5
```



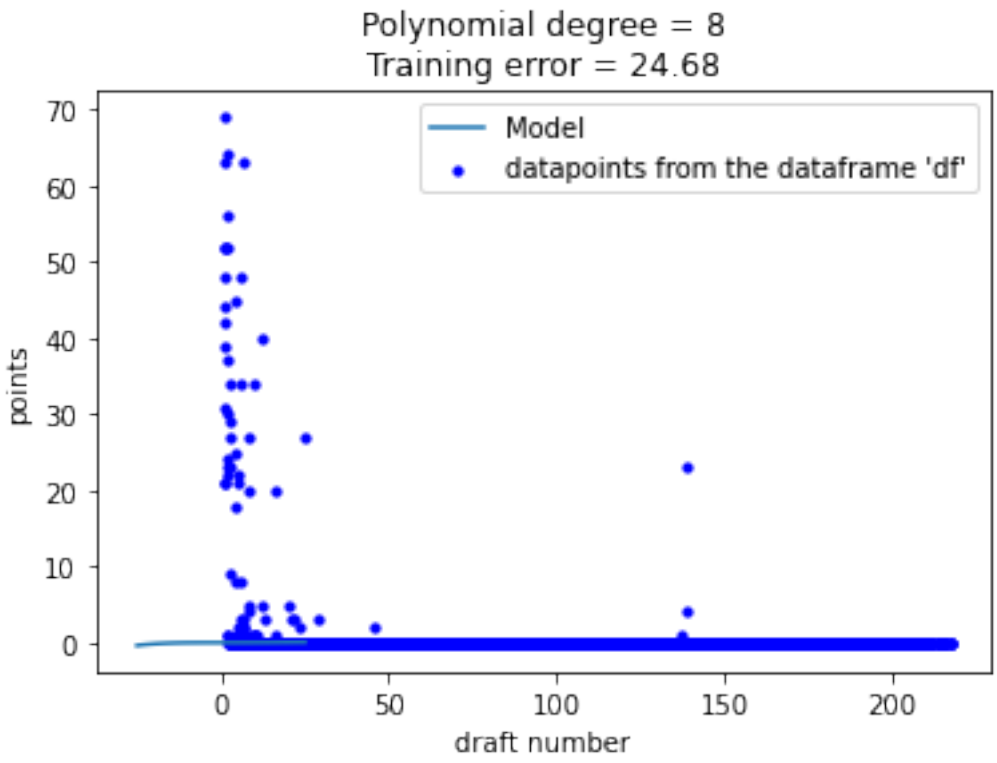

```
Polynomial degree =  6
```




```
Polynomial degree =  7
```




```
Polynomial degree =  8
```

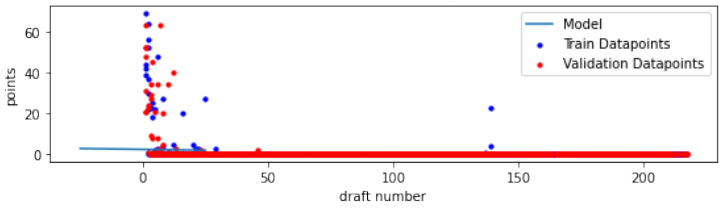


[9]:
```
# Split the dataset into a training set and a validation set like:
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=50)
```

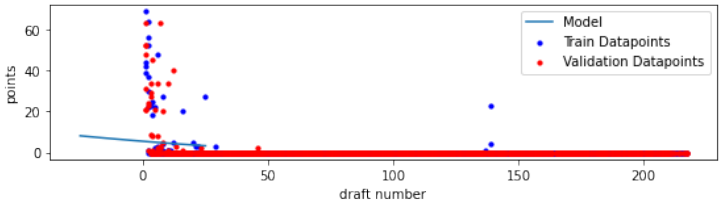
[10]:
```
## define a list of values for the maximum polynomial degree
degrees = [1,2,3,4,5,6,7]

# we will use this variable to store the resulting training errors for each␣
 ↪polynomial degree
tr_errors = []
val_errors = []

plt.figure(figsize=(8, 20))     # create a new figure with size 8*20
for i, degree in enumerate(degrees):     # use for-loop to fit polynomial␣
 ↪regression models with different degrees
    plt.subplot(len(degrees), 1, i + 1)     # choose the subplot

    lin_regr = LinearRegression(fit_intercept=False) # NOTE:␣
 ↪"fit_intercept=False" as we already have a constant iterm in the new feature␣
 ↪X_poly

    poly = PolynomialFeatures(degree=degree)     # generate polynomial features
```

```
    X_train_poly = poly.fit_transform(X_train)    # fit and transform the raw␣
↪features
    lin_regr.fit(X_train_poly, y_train)    # apply linear regression to these␣
↪new features and labels

    y_pred_train = lin_regr.predict(X_train_poly)    # predict values for the␣
↪training data using the linear model
    tr_error = (mean_squared_error(y_train,␣
↪y_pred_train),mean_absolute_error(y_train, y_pred_train))        # calculate␣
↪the training error
    X_val_poly =  poly.fit_transform(X_val)      # transform the raw features␣
↪for the validation data
    y_pred_val = lin_regr.predict(X_val_poly)      # predict values for the␣
↪validation data using the linear model
    val_error = (mean_squared_error(y_val, y_pred_val)␣
↪,mean_absolute_error(y_val, y_pred_val))       # calculate the validation␣
↪error



    tr_errors.append(tr_error)
    val_errors.append(val_error)
    X_fit = np.linspace(-25, 25, 100)    # generate samples
    plt.tight_layout()
    plt.plot(X_fit, lin_regr.predict(poly.transform(X_fit.reshape(-1, 1))),␣
↪label="Model")    # plot the polynomial regression model
    plt.scatter(X_train, y_train, color="b", s=10, label="Train Datapoints")    ␣
↪# plot a scatter plot of y(points) vs. X(draft number) with color 'blue' and␣
↪size '10'
    plt.scatter(X_val, y_val, color="r", s=10, label="Validation Datapoints")  ␣
↪ # do the same for validation data with color 'red'
    plt.xlabel('draft number')    # set the label for the x/y-axis
    plt.ylabel('points')
    plt.legend(loc="best")    # set the location of the legend
    plt.title(f'Polynomial degree = {degree}\nTraining errors: MSE={tr_error[0]:
↪.5}, MAE={tr_error[1]:.5}\nValidation errors: MSE={val_error[0]:.5},␣
↪MAE={val_error[1]:.5}')    # set the title
plt.show()    # show the plot

# sanity check the length of array tr_errors
assert len(tr_errors) == len(val_errors) == len(degrees)
```

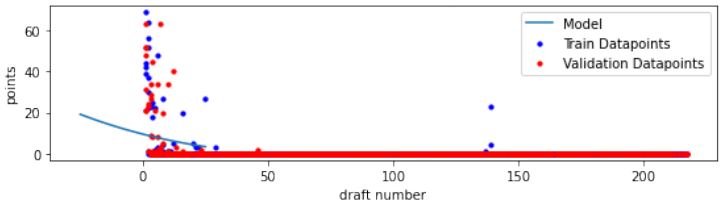
Polynomial degree = 1
Training errors: MSE=23.717, MAE=1.495
Validation errors: MSE=22.818, MAE=1.5536



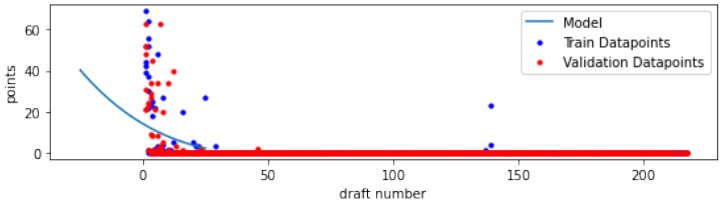
Polynomial degree = 2
Training errors: MSE=22.15, MAE=1.6249
Validation errors: MSE=21.435, MAE=1.73



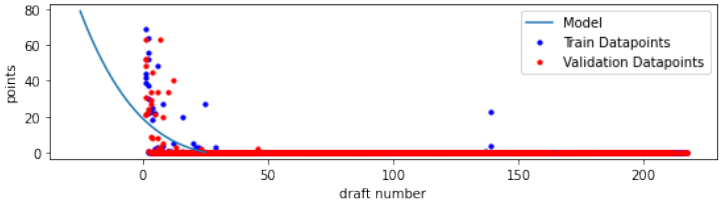
Polynomial degree = 3
Training errors: MSE=20.216, MAE=1.6803
Validation errors: MSE=19.888, MAE=1.7967



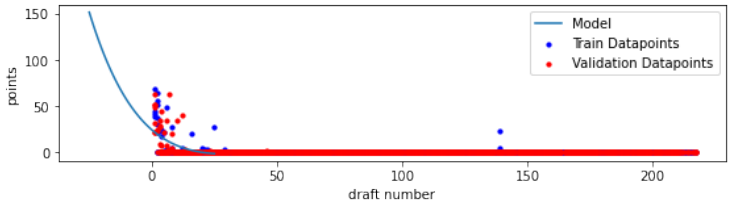
Polynomial degree = 4
Training errors: MSE=18.297, MAE=1.6454
Validation errors: MSE=18.379, MAE=1.7645



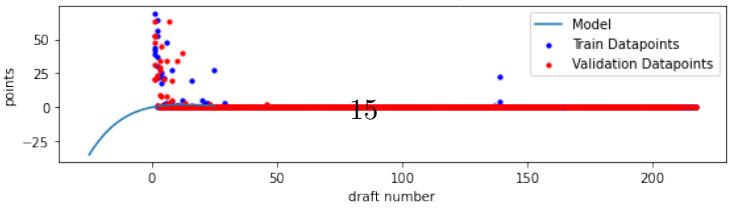
Polynomial degree = 5
Training errors: MSE=16.585, MAE=1.5442
Validation errors: MSE=17.041, MAE=1.6599



Polynomial degree = 6
Training errors: MSE=14.969, MAE=1.4394
Validation errors: MSE=15.974, MAE=1.5714



Polynomial degree = 7
Training errors: MSE=24.569, MAE=1.0031
Validation errors: MSE=23.488, MAE=1.0296



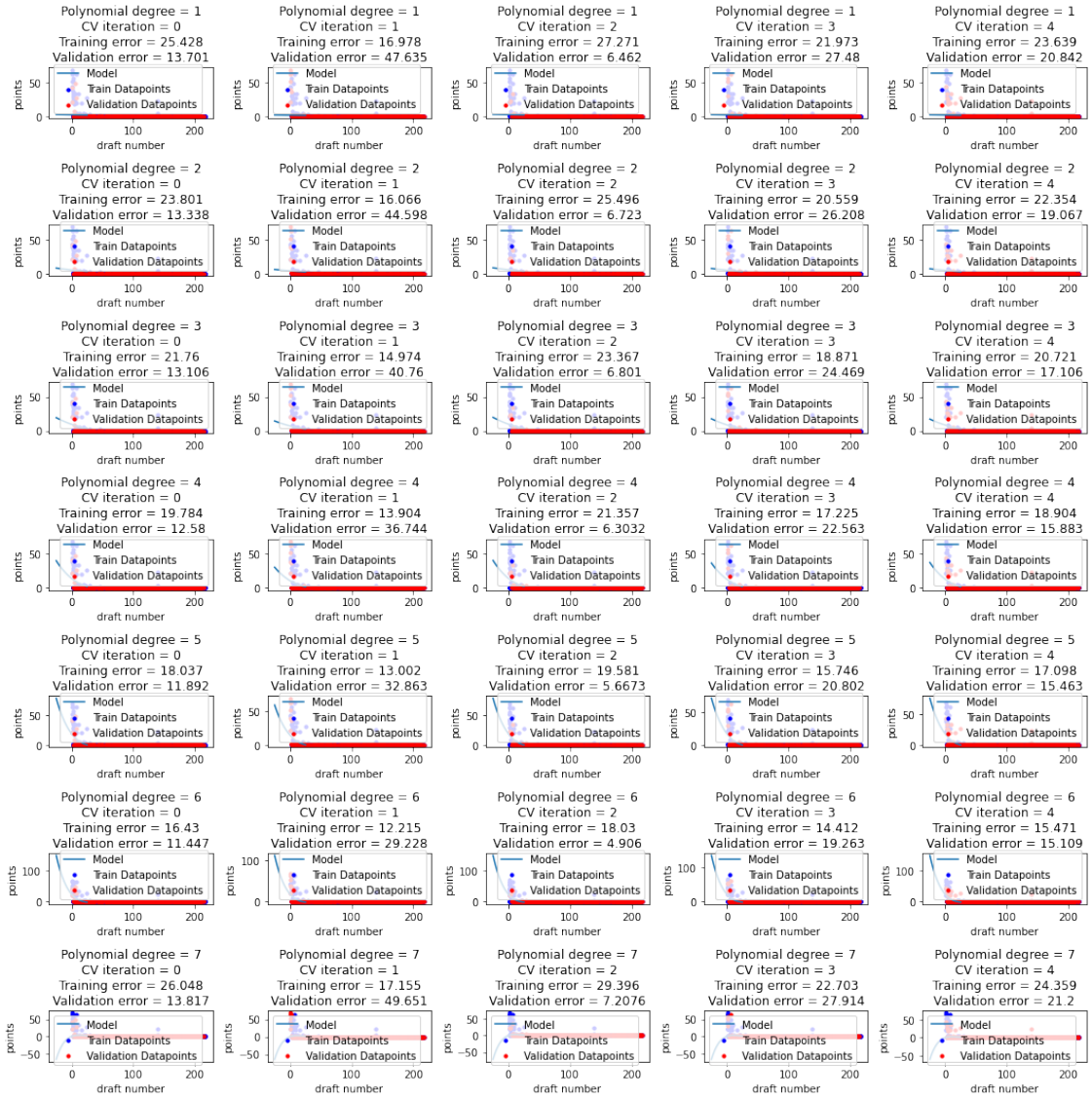
[11]:
```
X, X_test, y, y_test = train_test_split(X, y, test_size=0.2, random_state=50)
# 20% taken to test, others used for train/val data
```

[12]:
```
# Defining the kfold object we will use for cross validation
k, shuffle, seed = 5, True, 50
kfold = KFold(n_splits=k, shuffle=shuffle, random_state=seed)
```

[ ]:
```
```

[13]:
```
## define a list of values for the maximum polynomial degree
degrees = [1,2,3, 4, 5,6,7]

# we will use this variables to store the resulting training/validation errors␣
 ↪for each polynomial degree
# NB - this time we have multiple errors (for each CV step) for each degree, so␣
 ↪we store the errors in a dictionary
tr_errors = {}
val_errors = {}
test_errors = {}
plt.figure(figsize=(15, 15))    # create a new figure with size 8*20
for i, degree in enumerate(degrees):    # use for-loop to fit polynomial␣
 ↪regression models with different degrees
    tr_errors[degree] = [] # NB - now we will have k different errors per degree
    val_errors[degree] = []
    test_errors[degree] = []
    # We use the kfold object created earlier, to obtain train and validation␣
 ↪sets
    # for k iterations of training and evaluation
    for j, (train_indices, val_indices) in enumerate(kfold.split(X)):
        plt.subplot(len(degrees), k, i * k + j + 1)   # choose the subplot

        # Define the training and validation data using the indices returned by␣
 ↪kfold and numpy indexing

        X_train, y_train, X_val, y_val = X[train_indices], y[train_indices],␣
 ↪X[val_indices], y[val_indices]

        lin_regr = LinearRegression(fit_intercept=False) # NOTE:␣
 ↪"fit_intercept=False" as we already have a constant iterm in the new feature␣
 ↪X_poly
        poly = PolynomialFeatures(degree=degree)    # generate polynomial␣
 ↪features
        X_train_poly = poly.fit_transform(X_train)    # fit the raw features
```

```
        lin_regr.fit(X_train_poly, y_train)    # apply linear regression to␣
↪these new features and labels

        # Now we compute the errors on train and validation data obtained from␣
↪kfold
        y_pred_train = lin_regr.predict(X_train_poly)    # predict using the␣
↪linear model
        tr_error = mean_squared_error(y_train, y_pred_train)    # calculate the␣
↪training error
        X_val_poly = poly.transform(X_val) # fit the raw features for the␣
↪validation data
        y_pred_val = lin_regr.predict(X_val_poly) # predict labels for the␣
↪validation data using the linear model
        val_error = mean_squared_error(y_val, y_pred_val) # calculate the␣
↪validation error

        tr_errors[degree].append(tr_error) # # NB - We save all the errors to␣
↪analyze later
        val_errors[degree].append(val_error)
        X_fit = np.linspace(-25, 25, 100)    # generate samples
        plt.tight_layout()
        plt.plot(X_fit, lin_regr.predict(poly.transform(X_fit.reshape(-1, 1))),␣
↪label="Model")    # plot the polynomial regression model
        plt.scatter(X_train, y_train, color="b", s=10, label="Train␣
↪Datapoints")    # plot a scatter plot of y(points) vs. X(draft number) with␣
↪color 'blue' and size '10'
        plt.scatter(X_val, y_val, color="r", s=10, label="Validation␣
↪Datapoints")    # do the same for validation data with color 'red'
        plt.xlabel('draft number')    # set the label for the x/y-axis
        plt.ylabel('points')
        plt.legend(loc="best")    # set the location of the legend
        plt.title(f'Polynomial degree = {degree}\nCV iteration = {j}\nTraining␣
↪error = {tr_error:.5}\nValidation error = {val_error:.5}')    # set the title

    X_test_poly = poly.transform(X_test) # fit the raw features for the test␣
↪data
    y_pred_test = lin_regr.predict(X_test_poly) # predict labels for the test␣
↪data using the linear model
    test_error = mean_squared_error(y_test, y_pred_test) # calculate the test␣
↪error
    test_errors[degree].append(test_error)

plt.show()    # show the plot
```



[14]:
```
average_train_error, average_val_error = {}, {}
for degree in degrees:
    # Now we calculate the average train and validation errors for each␣
 ↪polynomial degree
    average_train_error[degree] = np.mean(tr_errors[degree])
    average_val_error[degree] = np.mean(val_errors[degree])

    print(f"Degree {degree}, avg train error = {average_train_error[degree]:.
 ↪5f}, "
          f"avg val error = {average_val_error[degree]:.5f}, " f"test error =␣
 ↪{test_errors[degree][0]:.5f}")
```

```
Degree 1, avg train error = 23.05769, avg val error = 23.22407, test error =
```

```
24.06858
Degree 2, avg train error = 21.65520, avg val error = 21.98704, test error =
22.19120
Degree 3, avg train error = 19.93868, avg val error = 20.44851, test error =
20.14778
Degree 4, avg train error = 18.23494, avg val error = 18.81470, test error =
18.20946
Degree 5, avg train error = 16.69276, avg val error = 17.33745, test error =
16.62991
Degree 6, avg train error = 15.31130, avg val error = 15.99061, test error =
15.33866
Degree 7, avg train error = 23.93214, avg val error = 23.95784, test error =
24.93285
```

[15]:
```
# Seems like the degree 6 is the best due to lowest error.
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: