Lecture 3 Code Workshop

19 February 2024

We will be running a Gaussian Process (GP) on our now well-known dataset to predict the yields of reactions.

Let's start by important the packages. We have almost the same packages as our PCA / SVM workshop last week, but instead of importing PCA and SVC, we are importing scikit's GaussianProcessRegressor. As a side note, there is also a GaussianProcessClassifier which is the GP model that one can use for classification. As we did classification last week, let's do regression this week. We also are importing kernels and the regression metrics R^2 score and MAE.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, RBF
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_absolute_error
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Since we talked about it in the lecture RMSE, I've also made a function that does just that.

In [2]:
```
def root_mean_squared_error(true, pred):
    mean_squared_error = np.mean((true - pred)**2)
    return (mean_squared_error)**(1/2)
```

We'll load our dataset as a pandas dataframe.

In [4]:
```
df = pd.read_csv('doyle.csv')
```

And we'll load in our reaction fingerprints to be our inputs for the same reasons we discussed in the Lecture 2 Workshop.

In [5]:
```
rxn_fps = np.load('rxn_fps_inputs.npy')
rxn_fps_inputs = StandardScaler().fit_transform(rxn_fps)
labels = np.array(df['yield'])
```

Similar to PCAs and SVMs, it's generally considered good practice to scale your inputs (reaction fingerprints). This is because GPs work closely with the variances of variables, which as we discussed in the previous workshop, are quite sensitive to scaling and units. We will see later on what happens if we don't scale our inputs. In this case our labels are the ***yields themselves*** - not a differentiation between good / bad yield.

We can define our GP. The kernel is an important parameter for all GP-based modelling. It (essentially) describes the kind of functions that we will be using to model the data - some kernels work better than others and its quite empirical what works best. You can find a list of available kernels from scikit here: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.gaussian_process.kernels.

In [3]:
```
# Define the GPs.
gp = GaussianProcessRegressor(kernel=RBF())
```

We will then split our data into training and testing sets.

```
In [6]: train_inputs, test_inputs, train_labels, test_labels = train_test_split(rxn_fps_inputs,
                                                                                labels,
                                                                                test_size=0.01,
                                                                                random_state=12)
        print(len(train_inputs),len(test_inputs))
```

```
4553 46
```

And train our GP.

```
In [7]: gp.fit(train_inputs, train_labels)
```

```
Out[7]: GaussianProcessRegressor(kernel=RBF(length_scale=1))
```

We can then predict on our held out test data, including the model's confidence (standard deviation).

```
In [8]: preds, std = gp.predict(test_inputs, return_std=True)
```

```
In [9]: print(std)
```

```
[0.75435213 0.88106746 0.63509824 0.70667784 0.55432507 0.97943236
 0.99994082 0.40976756 0.2552237  0.68689791 0.7546915  0.53644908
 1.         0.61700925 0.71624384 0.747511   0.38586958 0.35991362
 0.29971499 0.90858175 0.67194507 0.52280966 0.79919483 0.93080072
 0.51858171 0.47141973 0.45223563 0.53270947 0.83368558 0.91242318
 0.84562334 0.99722411 0.87727022 0.738838   0.84969692 0.21909667
 0.99978524 0.90296429 0.25090385 0.75474307 0.97100384 0.80580785
 0.83768031 0.54014315 0.8495193  0.74309062]
```

Notice that these standard deviations are scaled standard deviations - their values range between 0 and 1. To get the true standard deviation values, we have to un-scale them like so:

```
In [10]: scaled_std = std * np.std(train_labels) + np.mean(train_labels)
```

```
In [11]: print(scaled_std)
```

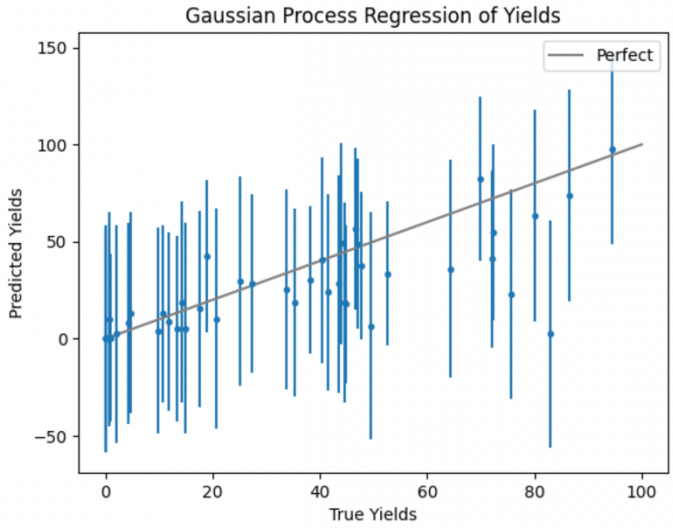
```
[51.79555605 55.31394172 48.48434536 50.47182897 46.24159241 58.04514742
 58.61458681 42.2277998  37.93672556 49.92261802 51.80497894 45.74524681
 58.61622988 47.98208542 50.7374391  51.60560487 41.56424702 40.84355227
 39.17207345 56.07790521 49.50743675 45.36653362 53.04066132 56.69483843
 45.24914029 43.93963775 43.40697092 45.64141251 53.99833344 56.18456652
 54.32979806 58.53915438 55.20850743 51.36478994 54.44290544 36.93362015
 58.61026671 55.92193053 37.81678043 51.80641095 57.81112084 53.22427868
 54.1092514  45.84781656 54.43797344 51.48286827]
```

We have pretty high levels of uncertainty across the board.

We can plot these points with their error bars to get a better feel of the predictions.

```
In [12]: plt.errorbar(test_labels, preds, yerr=scaled_std, fmt='.')
         plt.plot([0,100],[0,100], color='gray', label='Perfect')
         plt.title('Gaussian Process Regression of Yields')
         plt.xlabel('True Yields')
         plt.ylabel('Predicted Yields')
         plt.legend()
```

Out[12]: <matplotlib.legend.Legend at 0x7f8ed384bd50>



And we can also see their quantitative metrics.

```
In [32]: rmse = root_mean_squared_error(test_labels, preds)
         mae = mean_absolute_error(test_labels, preds)
         r2 = r2_score(test_labels, preds)

         print("RMSE:", rmse)
         print("MAE:", mae)
         print("R2", r2)
```

```
RMSE: 19.695679285710842
MAE: 12.199318930338391
R2 0.5043827475277524
```

Overall, it's not terrible but certainly not great.

Are the most confident predictions also the most accurate? We can check this by looking at the top 10 most confident (lowest standard deviation) predictions and comparing their metrics to the metrics above. Here are the standard deviations, sorted:

```
In [33]: # Top 10 most confident predictions.
         np.sort(scaled_std)
```

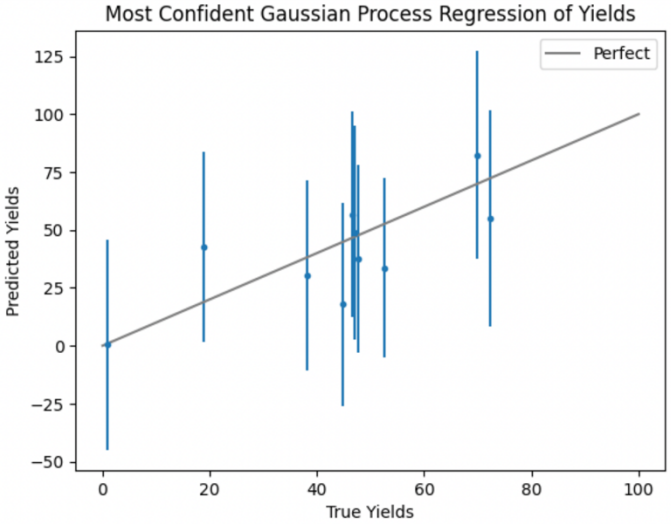
```
Out[33]: array([38.74609585, 40.56438902, 40.90207232, 40.9354263 , 43.66926307,
                44.54055415, 44.97687741, 45.43325598, 46.15576532, 46.88927048,
                47.03455449, 47.26427496, 47.38898102, 47.7661989 , 47.93638334,
                48.59727236, 49.12532608, 49.90728601, 50.38404842, 50.51806114,
                51.09572567, 51.1307312 , 51.1608422 , 51.23908033, 51.38166568,
                51.63073278, 51.72992493, 51.97972184, 52.58475645, 52.7768595 ,
                52.9337903 , 53.05886508, 53.16636318, 53.19137943, 54.15199746,
                54.37030502, 54.55910811, 54.84574349, 54.98472243, 55.16311216,
                56.87128582, 57.27909443, 57.88137686, 58.30409005, 58.43884431,
                58.61438321])
```

Let's find the most confident predictions and their associated ground truth values.

```
In [34]: threshold = np.sort(scaled_std)[9]
         confident_preds = []
         confident_stds = []
         confident_labels = []
         for p, s, l in zip(preds, scaled_std, test_labels):
             if s <= threshold:
                 confident_preds.append(p)
                 confident_stds.append(s)
                 confident_labels.append(l)
```

Now let's plot them.

In [35]:
```
plt.errorbar(confident_labels, confident_preds, yerr=confident_stds, fmt='.')
plt.plot([0,100],[0,100], color='gray', label='Perfect')
plt.title('Most Confident Gaussian Process Regression of Yields')
plt.xlabel('True Yields')
plt.ylabel('Predicted Yields')
plt.legend()
```



What about their metrics?

In [36]:
```
confident_labels = np.array(confident_labels)
confident_preds = np.array(confident_preds)
rmse = root_mean_squared_error(confident_labels, confident_preds)
mae = mean_absolute_error(confident_labels, confident_preds)
r2 = r2_score(confident_labels, confident_preds)

print("RMSE:", rmse)
print("MAE:", mae)
print("R2", r2)
```

```
RMSE: 15.421292056635998
MAE: 13.001308370554023
R2 0.4198870548765119
```

It seems that the most confident predictions do, in fact, line up with the higher accuracy predictions.

What happens if we change the kernel? Matern is another popular kernel so let's see how it fares when we ask those style of functions to be used in our GP modelling of reaction yields.

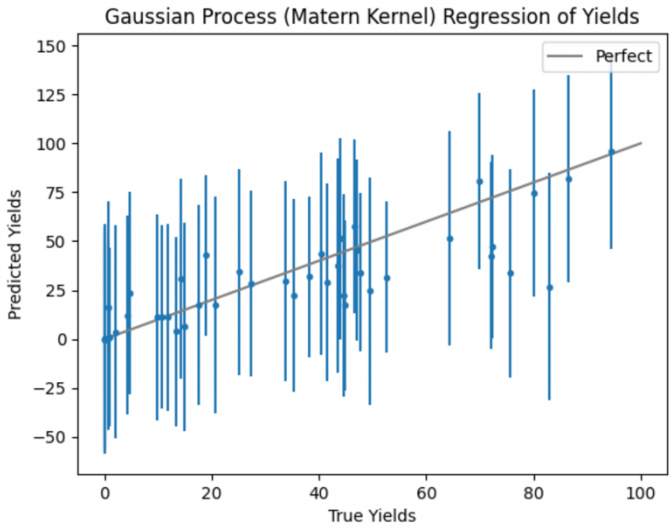
In [37]:
```
gp = GaussianProcessRegressor(kernel=Matern())
gp.fit(train_inputs, train_labels)
preds, std = gp.predict(test_inputs, return_std=True)
scaled_std = std * np.std(train_labels) + np.mean(train_labels)
rmse = root_mean_squared_error(test_labels, preds)
mae = mean_absolute_error(test_labels, preds)
r2 = r2_score(test_labels, preds)

print("RMSE:", rmse)
print("MAE:", mae)
print("R2", r2)
```

```
RMSE: 15.90102035508691
MAE: 10.460763587678503
R2 0.6769613585178806
```

It seems like it performs better than the RBF kernel, which makes sense. RBF chooses functions that are smooth and Matern has the GP choose functions that are more wiggly, which means the GP can model the weird activity cliffs a little easier.

Here's the plot of the Matern kernel predictions.

In [38]:
```
plt.errorbar(test_labels, preds, yerr=scaled_std, fmt='.')
plt.plot([0,100],[0,100], color='gray', label='Perfect')
plt.title('Gaussian Process (Matern Kernel) Regression of Yields')
plt.xlabel('True Yields')
plt.ylabel('Predicted Yields')
plt.legend()
```

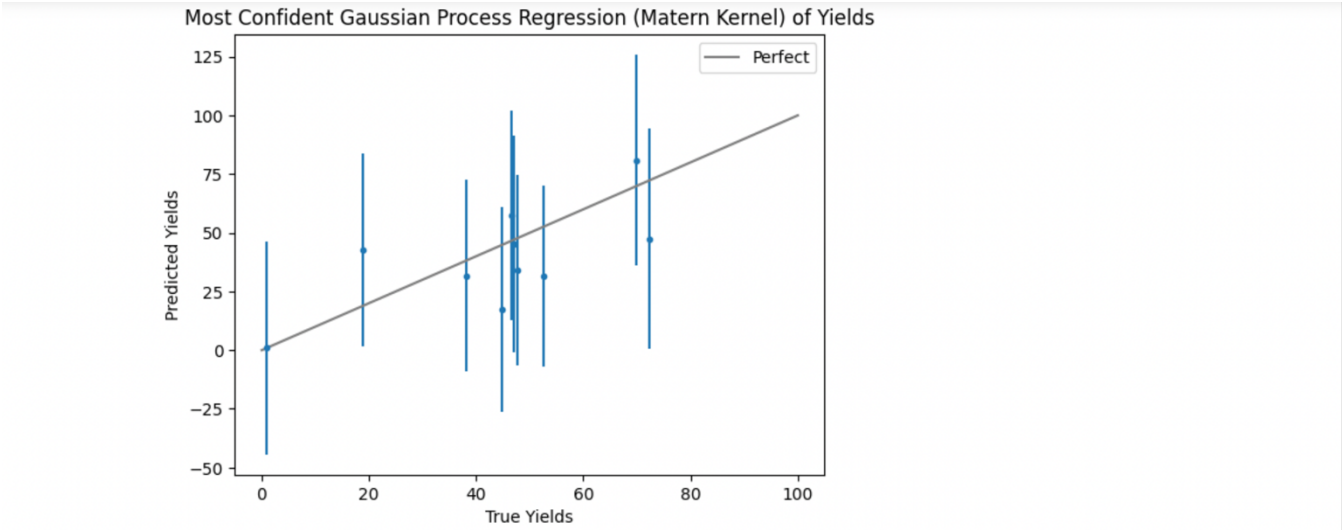


However, the top 10 most confident predictions are not, in this case, more accurate, so it's not always the case that the most confident predictions are the most accurate, especially when these predictions are so unconfident.

In [39]:
```
threshold = np.sort(scaled_std)[9]
confident_preds = []
confident_stds = []
confident_labels = []
for p, s, l in zip(preds, scaled_std, test_labels):
    if s <= threshold:
        confident_preds.append(p)
        confident_stds.append(s)
        confident_labels.append(l)

confident_labels = np.array(confident_labels)
confident_preds = np.array(confident_preds)
rmse = root_mean_squared_error(confident_labels, confident_preds)
mae = mean_absolute_error(confident_labels, confident_preds)
r2 = r2_score(confident_labels, confident_preds)

print("RMSE:", rmse)
print("MAE:", mae)
print("R2", r2)
```

```
RMSE: 16.926762574519493
MAE: 14.134777922609658
R2 0.3010938893213161
```

In [40]:
```
plt.errorbar(confident_labels, confident_preds, yerr=confident_stds, fmt='.')
plt.plot([0,100],[0,100], color='gray', label='Perfect')
plt.title('Most Confident Gaussian Process Regression (Matern Kernel) of Yields')
plt.xlabel('True Yields')
plt.ylabel('Predicted Yields')
plt.legend()
```



Finally, let's take a look at what happens if we don't scale our data.

```
In [24]: # Did not scale.
         gp = GaussianProcessRegressor(kernel=RBF())
         train_inputs, test_inputs, train_labels, test_labels = train_test_split(rxn_fps,
                                                                                 labels,
                                                                                 test_size=0.01,
                                                                                 random_state=12)
         gp.fit(train_inputs, train_labels)

         /Users/emmaking-smith/miniconda3/envs/ccdc3.7/lib/python3.7/site-packages/sklearn/gaussian_process/kernels.py:427:
         ConvergenceWarning: The optimal value found for dimension 0 of parameter length_scale is close to the specified low
         er bound 1e-05. Decreasing the bound and calling fit again may find a better value.
           ConvergenceWarning,

Out[24]: GaussianProcessRegressor(kernel=RBF(length_scale=1))
```

Well, we get a **warning**, that's what! Warnings are different from errors - it means that something may have gone wrong but not necessarily that the model was unable to run.

However, when we take a look at the predictions, we can see that the error was well warranted. Remember to scale your inputs.

```
In [26]: preds, std = gp.predict(test_inputs, return_std=True)
         preds

Out[26]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

– Until Next Time!