Lecture 2 Code Workshop

12 February 2024

We will start first by running a PCA of Abby Doyle's dataset, followed by an SVM classification of good yielding / poor yielding reactions. Like before, we begin by importing necessary packages to load the dataset, run mathematical operations, perform the modelling (sklearn), and visualize any results (matplotlib).

In [24]:
```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from rdkit import Chem
from sklearn.model_selection import train_test_split
from rdkit.Chem import AllChem
from sklearn.metrics import f1_score
import matplotlib.pyplot as plt
```

Let's load the dataset as a pandas dataframe.

In [2]:
```
df = pd.read_csv('doyle.csv')
df.head()
```

Out[2]:

	plate	row	col	base	base_cas_number	base_smiles	ligand	ligand_cas_number	ligand_smiles	aryl_halide_nu
0	1	1	1	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
1	1	1	2	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
2	1	1	3	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
3	1	1	4	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
4	1	1	5	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	

There are some functions that I have written to help expedite the process of setting up the inputs (Morgan fingerprints) for our models. It is not critical for this workshop to know what every line does. Briefly, make_single_fingerprint takes a single molecule to its corresponding Morgan fingerprint, make_inputs will make the inputs we need for the PCA and SVM, and classify_yields assigns a label of "1" if a reaction is high yielding and "0" if it is low yielding.

In [4]:
```
def make_single_fingerprint(smiles):
    mol = Chem.MolFromSmiles(smiles)
    fps = AllChem.GetMorganFingerprintAsBitVect(mol, 2, 1024)
    return np.array(fps)

def make_inputs(df, column_name):
    inputs = np.array([])
    df.loc[pd.isna(df[column_name]) == True, [column_name]] = ''
    for smiles in df[column_name]:
        fps = make_single_fingerprint(smiles)
        inputs = np.concatenate((inputs, fps))
    return inputs.reshape([len(df), -1])

def classify_yields(df):
    average_yield = np.mean(df['yield'])
    classified = []
    for y in df['yield']:
        if y >= average_yield:
            classified.append(1)
        else:
            classified.append(0)
    df['class_yield'] = classified
    return df
```

– Principal Component Analysis –

Let's first visualize what our chemical space looks in terms of reagents and reactants. We begin by defining our PCA, in this case a 2-dimensional PCA, which can be achieved by setting n_components=2.

In [3]:
```
pca = PCA(n_components=2)
```

We will then create our inputs for the PCA, which for this workshop are RDKit's Morgan fingerprints. The basic steps are to take a molecule in SMILES format to an RDKit molecule, and from an RDKit molecule to an RDKit fingerprint. These steps can be traced out in make_single_fingerprint().

For the sake of simplicity, I'm creating a new dataframe, which I'm calling pca_df, to keep track of the relevant PCA information. It will have the molecule SMILES and what kind of reaction component each molecule is.

In [13]:
```
pca_df = pd.DataFrame()
smiles = (df['aryl_halide_smiles'].unique().tolist() +
          df['ligand_smiles'].unique().tolist() + df['base_smiles'].unique().tolist() +
          df['additive_smiles'].unique().tolist())
labels = (['aryl halide'] * len(df['aryl_halide_smiles'].unique()) +
          ['ligand'] * len(df['ligand_smiles'].unique()) +
          ['base'] * len(df['base_smiles'].unique()) +
          ['additive'] * len(df['additive_smiles'].unique()))
```

```
In [16]: pca_df['smiles'] = smiles
         pca_df['labels'] = labels
         pca_df = pca_df.loc[pd.isna(pca_df['smiles']) == False]
```

```
In [17]: pca_df
```

Out[17]:

	smiles	labels
0	FC(F)(F)c1ccc(Cl)cc1	aryl halide
1	FC(F)(F)c1ccc(Br)cc1	aryl halide
2	FC(F)(F)c1ccc(I)cc1	aryl halide
3	COc1ccc(Cl)cc1	aryl halide
4	COc1ccc(Br)cc1	aryl halide
5	COc1ccc(I)cc1	aryl halide
6	CCc1ccc(Cl)cc1	aryl halide
7	CCc1ccc(Br)cc1	aryl halide
8	CCc1ccc(I)cc1	aryl halide
9	Clc1ccccn1	aryl halide
10	Brc1ccccn1	aryl halide

The basic steps for running a PCA is to:

1. Make and inputs. In this case, these are the Morgan fingerprints for each one of our reaction components.

```
In [19]: pca_inputs = make_inputs(pca_df, 'smiles')
         print(pca_inputs.shape)
         print(pca_inputs)
         print(pca_inputs[0][200:300])
```

```
(45, 1024)
[[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. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 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. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
```

2. Scale your inputs. Scaling is not always strictly necessary, but variables of different scale can seem like they have more variance (are very important), when in reality, their assay readouts may just be in a different unit / scale / etc. This is achieved by using the StandardScaler() function - you'll see this guy A LOT.

```
In [20]: # Running PCA
         pca_inputs = StandardScaler().fit_transform(pca_inputs)
         principal_components = pca.fit_transform(pca_inputs)
         principal_components
```

```
Out[20]: array([[-1.06822868,  3.48931189],
                [-1.11113462,  3.45337632],
                [-1.0985161 ,  3.52164038],
                [-0.55739449,  1.03123627],
                [-0.60030043,  0.9953007 ],
                [-0.58768192,  1.06356477],
                [-1.37683003,  1.18482841],
                [-1.41973597,  1.14889283],
                [-1.40711746,  1.2171569 ],
                [-0.31736485,  2.885326  ],
                [-1.35204128,  2.71535   ],
                [-1.00616315,  2.80067351],
                [-1.58802697,  3.41915153],
                [-1.6031811 ,  3.39893485],
                [-1.61440716,  3.44222418],
                [16.87234562,  2.35523373],
                [12.91016257,  0.85588299],
                [15.24045102, -2.09058786],
```

3. Then, run the PCA on the scaled inputs. This will give us our principal component 1 (first column) and principal component 2 (second column). I personally like to add this to our pca_df for ease of understanding.

```
In [21]: pca_df['PC1'] = principal_components[:, 0]
         pca_df['PC2'] = principal_components[:, 1]
         pca_df
```

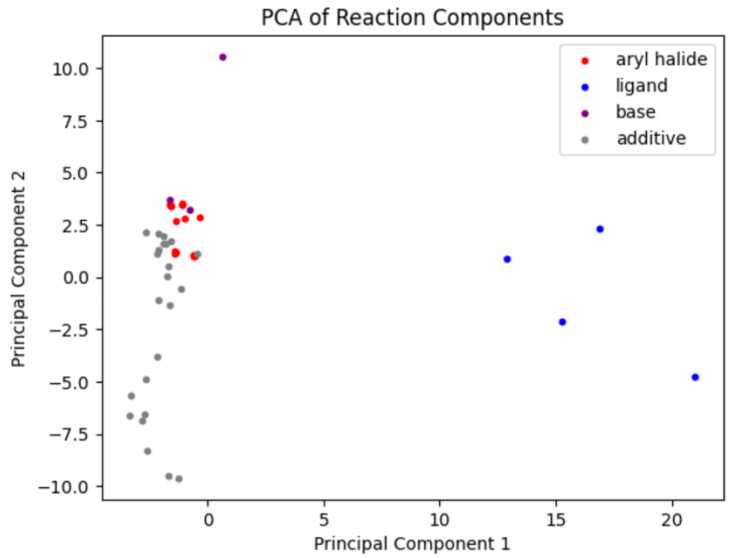
Out[21]:

	smiles	labels	PC1	PC2
0	FC(F)(F)c1ccc(Cl)cc1	aryl halide	-1.068229	3.489312
1	FC(F)(F)c1ccc(Br)cc1	aryl halide	-1.111135	3.453376
2	FC(F)(F)c1ccc(I)cc1	aryl halide	-1.098516	3.521640
3	COc1ccc(Cl)cc1	aryl halide	-0.557394	1.031236
4	COc1ccc(Br)cc1	aryl halide	-0.600300	0.995301
5	COc1ccc(I)cc1	aryl halide	-0.587682	1.063565
6	CCc1ccc(Cl)cc1	aryl halide	-1.376830	1.184828
7	CCc1ccc(Br)cc1	aryl halide	-1.419736	1.148893
8	CCc1ccc(I)cc1	aryl halide	-1.407117	1.217157
9	Clc1ccccn1	aryl halide	-0.317365	2.885326
10	Brc1ccccn1	aryl halide	-1.352041	2.715350

3. Visualize the PCA with plotting software - color coding is essential to drawing conclusions. **NOTE:** Plotting software is pretty unintuitive for all plotting packages, so don't worry about each line of code.

```
In [28]: # Plotting
colors = ['red', 'blue', 'purple', 'gray']
for label, color in zip(pca_df['labels'].unique(), colors):
    idxs = pca_df.loc[pca_df['labels'] == label].index
    plt.scatter(pca_df.loc[idxs, 'PC1'], pca_df.loc[idxs, 'PC2'], s=10, c=color, label=label)
plt.title('PCA of Reaction Components')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
```

```
Out[28]: <matplotlib.legend.Legend at 0x7fe98fb319d0>
```



Here we can see the distribution of reactants and reagents. Ligands are pretty different from everything else and we have a good mix of additives. Other than that, not super interesting but a good warm up.

We can also visualize each **reaction** (via reaction fingerprints) color coded by its corresponding yield. The reaction fingerprints (https://rxn4chemistry.github.io/rxnfp/) were constructed previously by me as they can take some time to create, but the functions that I wrote to do so are as follows.

```
##### Make Rxn Fingerprint Functions ######
def make_rxn_fp(rxns):
    model, tokenizer = get_default_model_and_tokenizer()
    rxnfp_generator = RXNBERTFingerprintGenerator(model, tokenizer)
    rxn_fps = []
    for rxn in rxns:
        rxn_fps.append(rxnfp_generator.convert(rxn))
    return np.array(rxn_fps).reshape([len(rxns), -1])
```

```
def make_rxn(row):
    if row['product_smiles'] == '':
        rxn = row['ligand_smiles'] + '.' + row['additive_smiles'] + '.' + row['base_smiles'] + '>>' + row['ligand_sm
    else:
        rxn = row['aryl_halide_smiles'] + '.' + row['ligand_smiles'] + '.' + row['additive_smiles'] +  '.' + row['ba
    rxn = rxn.replace('..', '')
    if rxn[0] == '.':
        rxn = rxn[1:]
    return rxn
```

```
def make_rxn_inputs(df):
    df.loc[pd.isna(df['aryl_halide_smiles']) == True, ['aryl_halide_smiles']] = ''
    df.loc[pd.isna(df['ligand_smiles']) == True, ['ligand_smiles']] = ''
    df.loc[pd.isna(df['additive_smiles']) == True, ['additive_smiles']] = ''
    df.loc[pd.isna(df['base_smiles']) == True, ['base_smiles']] = ''
    df.loc[pd.isna(df['product_smiles']) == True, ['product_smiles']] = ''
    rxns = []
    for i in range(len(df)):
        r = make_rxn(df.iloc[i])
        rxns.append(r)

    rxn_inputs = make_rxn_fp(rxns)
    return rxn_inputs
```

Each row in our dataframe will be converted to its reaction string (each reactant's SMILES separate by a "." and then ">>" will be added after all the reactants followed by the product SMILES). This reaction string will be converted to a vector using a pre-made model.

To save time, I have saved these reaction fingerprints so that I can load them in quickly. This is done as follows.

```
In [30]: rxn_fps = np.load('rxn_fps_inputs.npy')
         print(rxn_fps.shape)
         rxn_fps

         (4599, 256)

Out[30]: array([[-0.34900159, -0.48079336,  0.16739927, ...,  0.77806962,
                  -2.06544399,  0.58454704],
                 [-0.54790264, -0.28887191,  0.2148052 , ...,  0.66739988,
                  -2.2874527 ,  0.49061108],
                 [-0.46584156, -0.30957404,  0.4540565 , ...,  0.51171994,
                  -2.20896935,  0.75939816],
                 ...,
                 [-0.54888171, -0.20472461,  0.48514891, ...,  0.9333204 ,
                  -2.06480026,  0.14035255],
                 [-0.53889406, -0.34741053,  0.63243479, ...,  0.63681674,
                  -2.17075968,  0.6155014 ],
                 [-1.72162199, -1.7136091 , -0.82381564, ...,  1.65698242,
                  -0.2131172 , -0.07295071]])
```

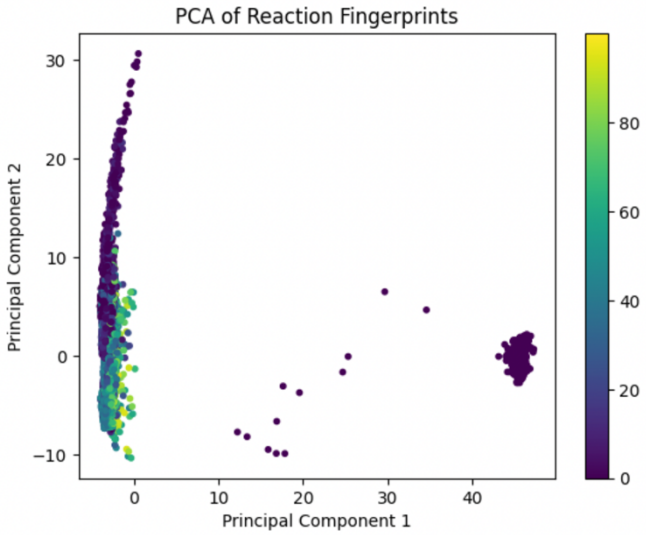
We will run through the same steps again. Scale, run the PCA, and visualize.

```
In [31]: # Run PCA on rxn fingerprints.
rxn_fps_inputs = StandardScaler().fit_transform(rxn_fps)
principal_components_rxn = pca.fit_transform(rxn_fps_inputs)
df['PC1'] = principal_components_rxn[:, 0]
df['PC2'] = principal_components_rxn[:, 1]
df
```

l_halide_number	aryl_halide	aryl_halide_smiles	additive_number	additive	additive_smiles	product_smiles	yield	PC1	PC2
1.0	1-chloro-4-(trifluoromethyl)benzene	FC(F)(F)c1ccc(Cl)cc1	NaN	NaN	NaN	Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1	26.888615	-3.643722	1.651305
2.0	1-bromo-4-(trifluoromethyl)benzene	FC(F)(F)c1ccc(Br)cc1	NaN	NaN	NaN	Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1	24.063224	-3.373910	-3.870098
3.0	1-iodo-4-(trifluoromethyl)benzene	FC(F)(F)c1ccc(I)cc1	NaN	NaN	NaN	Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1	47.515821	-3.354322	-5.993568
4.0	1-chloro-4-methoxybenzene	COc1ccc(Cl)cc1	NaN	NaN	NaN	COc1ccc(Nc2ccc(C)cc2)cc1	2.126831	-3.804392	4.344261
5.0	1-bromo-4-methoxybenzene	COc1ccc(Br)cc1	NaN	NaN	NaN	COc1ccc(Nc2ccc(C)cc2)cc1	47.586354	-3.457316	-3.103531

This time for visualization, we will add a term called a "cmap" which allows us to have a gradient color bar rather than individual colors. We can now see reactions that performed very well and reactions that were lower yielding.

```
In [39]: # Plotting
plt.scatter('PC1', 'PC2', data=df, c='yield', cmap='viridis', s=10)
plt.title('PCA of Reaction Fingerprints')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar()
```



– Support Vector Machine Classifier –

Moving on to the SVM classifier. I want to be able to classify a reaction as good yielding (above the mean yield) or poor yielding (below the mean yield). It may be tempting to simply create molecular fingerprints for one component (starting material, product, additive, etc.) and use those fingerprints as inputs to the SVM classifier but this is the wrong way to go about it.

Let's take product as an example. Take a look at the number of unique products we have (6, including reactions that led to no product). And how many reactions we have in the entire dataset (4,599).

```
In [44]: # Products' Morgan fingerprints as inputs?
         print(df['product_smiles'].unique())
         print(len(df))
```

```
['Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1' 'COc1ccc(Nc2ccc(C)cc2)cc1'
 'CCc1ccc(Nc2ccc(C)cc2)cc1' 'Cc1ccc(Nc2ccccn2)cc1' 'Cc1ccc(Nc2cccnc2)cc1'
 nan]
4599
```

If we featurize our reactions solely by the product, we will end up in a situation where one input will have two or more different outputs. For example, the product, "Cc1cc(Nc2ccc(C(F)(F)F)cc2)cc1", has yields ranging from 0% to 55%. **This is very difficult to model!** The entire premise of machine learning (and mathematical functions) is that the same input will always give you the same output. Having 2+ possible, correct, yields for one input violates this.

```
In [47]: np.sort(df.loc[df['product_smiles'] == 'Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1', ['yield']].values.reshape([-1]))
```

```
Out[47]: array([ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
                 0.        ,  0.        ,  0.        ,  0.29652464,  0.2991536 ,
                 0.32393268,  0.33050129,  0.4138277 ,  0.41961234,  0.43228279,
                 0.44913528,  0.52226731,  0.54679138,  0.55204991,  0.66656148,
                 0.69831869,  0.70155223,  0.70766924,  0.72679069,  0.74741177,
                 0.75751912,  0.79061519,  0.79710472,  0.80567729,  0.87182075,
                 0.90886504,  0.92760453,  1.00926941,  1.03936431,  1.06906071,
                 1.12036933,  1.15939127,  1.22290693,  1.23263237,  1.24900846,
                 1.25144582,  1.25250533,  1.26109186,  1.2978186 ,  1.46454789,
                 1.63782786,  1.71151259,  1.78497243,  1.78558756,  1.83135149,
                 1.89618159,  1.91824842,  1.9832045 ,  2.00870867,  2.03885337,
                 2.06909823,  2.10360729,  2.10614363,  2.25635985,  2.39187081,
                 2.44310653,  2.50548932,  2.51051068,  2.5264192 ,  2.550915  ,
                 2.5656045 ,  2.74121661,  2.74907447,  2.82873897,  2.83256363,
                 2.87467931,  2.8878154 ,  2.93385881,  2.96886976,  2.97486192,
                 3.07461208,  3.10979023,  3.14193466,  3.15727363,  3.16019372,
```

```
In [47]: np.sort(df.loc[df['product_smiles'] == 'Cc1ccc(Nc2ccc(C(F)(F)F)cc2)cc1', ['yield']].values.reshape([-1]))
```

```
                 44.65371267, 44.68109157, 44.71241275, 44.78270058, 44.81754321,
                 45.01345928, 45.04099086, 45.04873884, 45.08048043, 45.15638688,
                 45.20399384, 45.52244321, 45.77926691, 45.78100103, 45.87563784,
                 45.90521481, 45.90687609, 45.92980056, 45.96047749, 46.14092963,
                 46.24961136, 46.25838016, 46.27953078, 46.36906778, 46.63249784,
                 46.6498955 , 46.6599306 , 46.67923287, 46.74711139, 46.76206263,
                 46.79588748, 47.02427089, 47.05511825, 47.07201706, 47.09445645,
                 47.09764929, 47.10591975, 47.11942721, 47.13760632, 47.29404915,
                 47.42870408, 47.45149232, 47.51582059, 47.555818  , 47.5989822 ,
                 47.67937659, 47.94242943, 48.00212967, 48.06786456, 48.32439551,
                 48.38997285, 48.4805474 , 48.49744103, 48.56121582, 48.69500261,
                 48.86581696, 49.15372199, 49.30223849, 49.46796695, 49.51819322,
                 49.58296909, 49.6136763 , 50.12467654, 50.13634297, 50.27132659,
                 50.28665363, 50.46562134, 50.58289676, 50.86308576, 51.07128287,
                 51.11682644, 51.209226  , 51.39708789, 51.75674767, 52.01896484,
                 52.13571621, 52.24472601, 52.3137348 , 52.37811419, 52.49415636,
                 52.55061961, 52.58811564, 52.67302388, 53.49467932, 53.59986981,
                 53.83922891, 53.85457225, 53.92524684, 54.22375463, 54.24578847,
                 55.16379105, 55.56585889])
```

Thus, to accurately model this data, it's better to featurize the **reaction** not any **individual component**. But don't just take my word for it - let's try it! Let's run an SVM classifier by predicting high yielding / low yielding reactions from just the product identity and see how we do. Then we can compare it to an SVM classifier trained on the previously made reaction fingerprints.

The steps to running an SVM are:

1. Define your SVM.

```
In [40]: # SVM Classifier.
         svm = SVC()
```

2. Create your inputs. In this case, we are using just the products' Morgan fingerprints.

In [48]:
```
product_inputs = make_inputs(df, 'product_smiles')
```

In [50]:
```
print(product_inputs.shape)
print(product_inputs)
print(product_inputs[0][200:300])
```

```
(4599, 1024)
[[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. 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. 1. 1. 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. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
```

3. Scale your inputs.

In [64]:
```
product_inputs = StandardScaler().fit_transform(product_inputs)
```

4. Create your labels.

In [51]:
```
print(np.mean(df['yield']))
df = classify_yields(df)
df[['yield', 'class_yield']]
```

```
30.86856260493172
```

Out[51]:

	yield	class_yield
0	26.888615	0
1	24.063224	0
2	47.515821	1
3	2.126831	0
4	47.586354	1
...	...	...
4594	86.233157	1
4595	1.440081	0
4596	43.538365	1
4597	69.795902	1
4598	0.000000	0

4599 rows × 2 columns

In [52]:
```
labels = np.array(df['class_yield'])
```

5. Split the inputs and labels into training and testing sets.

In [65]:
```
train_inputs, test_inputs, train_labels, test_labels = train_test_split(product_inputs,
                                                                        labels,
                                                                        test_size=0.33,
                                                                        random_state=12)

print(train_inputs.shape, test_inputs.shape, train_labels.shape, test_labels.shape)
```

```
(3081, 1024) (1518, 1024) (3081,) (1518,)
```

3. Fit the SVM on the training data.

In [66]:
```
# Fit the SVM
svm.fit(train_inputs, train_labels)
```

Out[66]:
```
SVC()
```

4. Predict the values for the testing data.

```
In [67]: preds = svm.predict(test_inputs)
```

5. See the evaluation score (we are using F-scores).

```
In [68]: product_only_score = f1_score(test_labels, preds)
         product_only_score
Out[68]: 0.3942505133470226
```

Kind of marginal F-score. Now let's compare that to the SVM classifier on the reaction fingerprints that we used for our PCA.

```
In [60]: # Rxn fingerprints as inputs to SVM.
         train_inputs, test_inputs, train_labels, test_labels = train_test_split(rxn_fps_inputs,
                                                                                 labels,
                                                                                 test_size=0.33,
                                                                                 random_state=12)
In [61]: svm.fit(train_inputs, train_labels)
Out[61]: SVC()

In [62]: preds = svm.predict(test_inputs)

In [63]: rxn_fps_score = f1_score(test_labels, preds)
         rxn_fps_score
Out[63]: 0.792507204610951
```

That's significantly better!

– Until Next Time!