STA 561

Final Project

Donald Cayton[1], John Gillen[1], Joanna Peng[2], Sayali Pingle[1], Juanita Pombo Garcia[3]

Affiliations

Department of Statistics, Duke University, Durham, NC, 27705, USA.

Department of Biomedical Engineering, Duke University, Durham, NC, 27705, USA.

Department of Mechanical Engineering and Materials Science, Duke University, Durham, NC, 27705, USA.

Contributions

DC and SP developed the machine learning algorithm.

JG prepared the technical details and preliminary results section.

JP prepared and simulated the data.

JP and JPG prepared the press release and FAQ section.

Press Release

FOR IMMEDIATE RELEASE

Automating Directed Evolution: Enabling Hypermutation of Multiple Genes of Interest in Parallel for High-Throughput Maturation of Protein Performance

April 26, 2024, Durham, North Carolina – Four enthusiastic Duke students have combined their passion in protein engineering and machine learning (ML) in an interdisciplinary project that enables the automation and parallelization of the Directed Evolution (DE) of proteins. The group ingeniously merged existing mechanical and electronic technologies employed by research labs worldwide with an in-house developed ML algorithm, to present the research community a robot capable of performing *in vivo* DE cycles autonomously and adaptively.

This robot operates within a novel and growing field of research, having been awarded the Nobel Prize in chemistry in 2018. In a nutshell, DE processes apply the principles of natural selection in a tunable and accelerated fashion to “develop new types of chemicals for the greatest benefit of humankind”, said Claes Gustafsson, Nobel Committee for Chemistry. Some examples of new types of chemicals achieved through DE include higher-performing biological drugs and laboratory reagents.

While DE methods vary, they typically follow similar cyclic workflows (Figure 1A) consisting of performing high-error polymerase chain reaction (PCR) on your gene of interest (GOI) to create a library of mutant clones; using viruses to insert the GOI library into cells (which are capable of expressing the GOI’s protein); selecting a subset of cells that correspond to high GOI performance; and extracting the DNA of this subset for the next cycle of DE. “Despite the advantages DE brings to the field of protein engineering, these classical methods suffer low-throughput and lack of scalability due to its manual, labor-intensive workflow”, says Joanna Peng, senior student at Duke University and the group’s domain expert. Recently, improved DE approaches have arisen where the diversification/hypermutation of a given GOI could occur completely *in vivo. In vivo* hypermutation more closely replicates natural evolutionary processes while simultaneously cutting the amount of manual work needed to complete a DE cycle by half

by allowing researchers to bypass the PCR and imperfect DNA isolation steps of classical DE methods (Figure 1B).

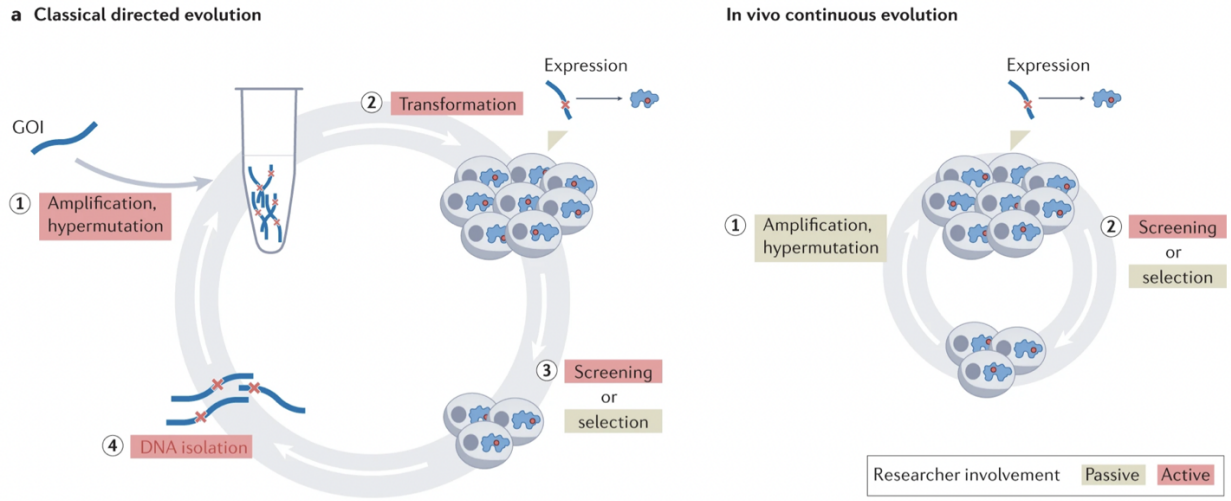


Figure 1. An example of the workflow of a single cycle of (A) classical and (B) in vivo directed evolution (DE). Notably, in vivo DE reduces the workflow in half, bypassing the transformation (PRC) and DNA isolation steps. [2]

Despite these improvements to what are considered gold-standard DE techniques, *in vivo* DE workflows still suffer from lack of scalability as GOIs must be evolved one at a time with each cell-sorting and selection experiment requiring 8-12 hours of manual monitoring. Moreover, the latter step (screening or selection) involves isolating the highest performing cell population to carry the next cycle of DE on, which is typically hand-picked by the experts. This process is known as gating, and it consists of hand-drawing a trapezoid-shaped gate on the GOI-protein binding versus GOI expression plot, that simultaneously maximizes both parameters, as shown in Figure 2. The exact shape of the trapezoid gate depends on the DE cycle number, strength in fluorescence, and a principle known as the Avidity Effect, requiring the input of a domain expert in every DE cycle. This further hinders the scalability and automation of *in vivo* DE methods.

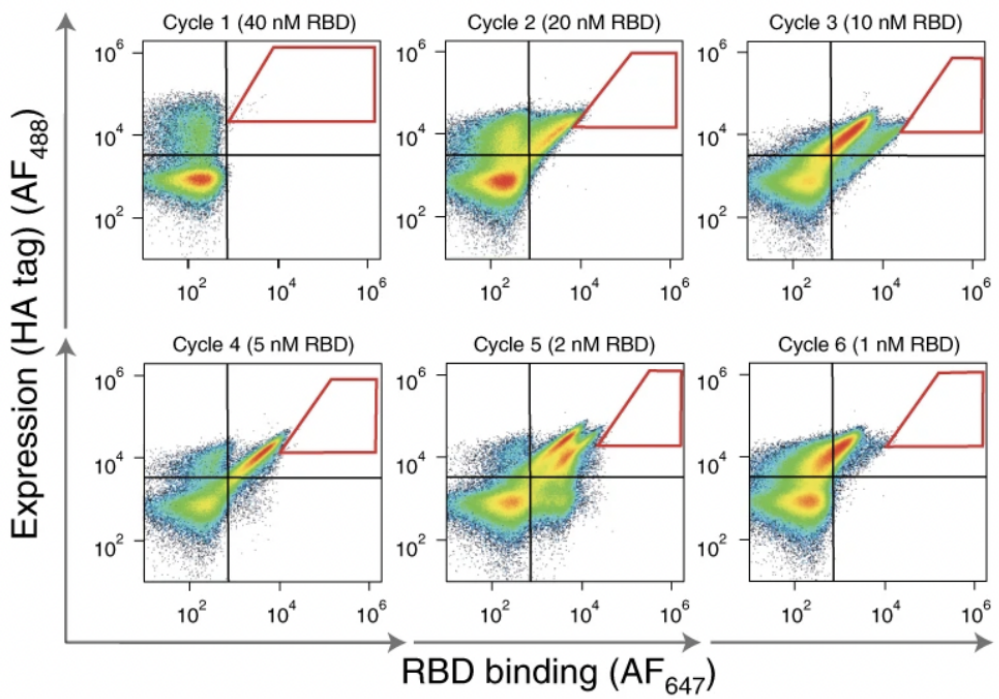


Figure 2. Visualizing results of six DE cycles with flow cytometry plots (GOI expression v. GOI performance). The red trapezoid indicates the manually drawn selection gate to indicate which cells were chosen to continue evolving. [5]

To get around this hurdle, the group has devised an ML classification approach to automate the gating process employing ample training data produced by domain experts. This software is incorporated within a robot, consisting of a smart bioreactor, a centrifuge, and a fluorescence-assisted cell sorting (FACS) device. The smart bioreactor regularly monitors the growth phase of the cell culture by measuring optical density with a spectrophotometer. The centrifuge separates the cells from their media to be treated with fluorescent labels needed during selection. Lastly, the FACS device quantifies GOI/protein performance and subsequently separates a fraction of the highest-performing clones for continued evolution in subsequent DE cycles. FACS data is fed into the ML algorithm, which carries out the cycle-dependent selection of the subset of clones to continue evolving. Because robots are indifferent toward labor-intensive tasks, this approach allows for the simultaneous evolution of multiple GOI clones, or unique GOI themselves.

Remarkably, the group presents to the research community a robot that serves as a platform to conduct DE experiments automatically and in parallel, increasing the scale of DE experiments and potentially speeding up the generation of protein-based drugs or

reagents. “It is always exciting to see how we can enhance and streamline already promising scientific methods by incorporating machine learning”, said Juanita Pombo, a member of the group and Ph.D. student in mechanical engineering and materials science.

FAQs

Q1: What is Directed Evolution?

Natural selection is an evolutionary phenomenon where inevitable errors in DNA replication and their resulting mutations yield new phenotypes that can be favorable or deleterious to survival. This process of toggling between mutation and selection is a major contributor to the immense biodiversity seen today. While impressive, these evolutionary processes operate on timescales of millions or even billions of years. To capitalize on the biology-enhancing capacity of evolutionary processes, Directed Evolution (DE) uses tunable, synthetic means of mutating and selecting superior phenotypes at an accelerated timescale of days. DE has been applied to a variety of protein encoding genes; a couple notable examples include the generation of ultrapotent neutralizing antibodies against the coronavirus spike protein and the development of novel enzymes that operate orthogonally to native systems.

Q2: What types of Gene of Interests (GOI) or proteins is this robot compatible with?

A chosen GOI can encode any type of protein (binders, enzymes, etc.) so long that the performance can be measured with fluorescence. In discussing many of the technical details below, we will use the example of evolving an ultrasensitive binder.

Q3: What is Autonomous Hypermutation Yeast Surface Display (AHEAD)?

AHEAD is a system or platform for *in vivo* continuous directed evolution. These synthetically engineered yeast cells utilize an error-prone polymerase with a rapid error rate of 10^{-5} errors/base to mutate and diversify a GOI. Orthogonal to the error prone polymerase is *S. cerevisiae’s* native machinery that makes sure the genetic code needed for growth, replication, and overall survival is being replicated normally. These cells are genetically programmed to grow and divide (and therefore mutate) when in glucose-based media and to produce/express the GOI when in galactose-based media (a step known as induction). By cycling through stages of growth/mutation and selection/cell-sorting this system facilitates directed evolution in a highly controlled manner across a more vast fitness landscape.

Q4: How will the robot perform *in vivo* continuous directed evolution autonomously?

For an evolving population of cells, the robot will monitor optical density (OD), which is a well-established measurement for the population's growth phase, to know when to pass cells into different culture media. For example, glucose-based media is used for growth, and typically cells are cultured till at least an OD of 0.8 before they are passed into galactose-based media, which is used to induce gene expression of the GOI. After passing the population of cells into the induction (galactose) media, cells will be given a standard 24 hours to express the GOI before selection occurs. Note that through every transcription of the DNA, and therefore every cell division, mutation and diversification of the GOI is occurring. Thus, each cell harbors a unique mutant clone of the GOI.

Once the cells are properly grown and induced, the performance of each mutant GOI clone will be quantified through a fluorescently-labeled antibody (provided by the user). Since every cell expresses variable amounts of the GOI, another fluorescently-labeled antibody will be used to quantify the GOI expression level. This quantity is necessary for normalizing mutant GOI performance readings because higher levels of protein expression artificially increases the protein performance readings. By normalizing by expression, we are able to select cells that harbor the best mutants of a GOI rather than cells that simply have higher levels of protein expression. When each cell within the induced population has their expression and performance quantified by the two fluorescently-labeled antibodies, these values are passed into our selection algorithm (technical details described below) to determine whether that cell should be discarded or selected to continue evolving in subsequent rounds. The selection algorithm will work in tandem with a cell sorter to physically separate the cells that demonstrate "superior fitness" from those with less impressive performance. Cells that are selected to continue evolving are returned to glucose media for continued growth/diversification.

Example of evolving a binder: Cells can be encoded to express a parent binder on their surface. After the appropriate amount of growth and induction, these cells would be incubated with the target protein at variable concentrations to empirically determine the best selection conditions. A fluorescent antibody against the target protein can directly measure the affinity of an individual cell's encoded binder for the target protein. Across the rounds of DE, the concentration of target protein incubated with the cells prior to selection is decreased to increase selection pressure. A good selection pressure is typically when only 5-30% of the cell population is capable of binding (a lower percentage would equate to a more aggressive selection strategy). The top 5% or so of these binders are selected to proceed to the subsequent round of DE.

Q5: How will the robot know when to terminate the directed evolution cycles?

As discussed earlier, the selection pressure applied in each subsequent round of DE becomes more aggressive (in our binder example, the concentration of target protein incubated with the cells decreases each round). Thus, a user can input a desired fold change in the final selection pressure to indicate when the robot should terminate performing DE cycles. If the population of evolving cells is incapable of reaching the user-

inputted fold change due to hitting a limit in the protein optimization (similar to falling into a local or global minima during ML optimization where improvement stalls to insignificant levels), the robot will be programmed to auto-terminate and notify the user.

Q6: Does the necessary hardware exist to accompany the algorithm described here?

Yes, all the necessary hardware/technology to develop the proposed robot exists, but they will require software integration, allowing the different components to function together. Bioreactor technology that autonomously monitors cell growth through OD and adds media when needed is well-established and widely used in industry. The selection component of our robot would require a flow cytometer (an instrument capable of reading fluorescence on a per cell basis) and a cell sorter (takes fluorescent readings of individual cells and parses them into their user-designated buckets). Both these technologies are prolifically used by today's scientific community. Since all the key pieces of hardware are well-established, we believe building a robot in-house that integrates these technologies is well within our capacity.

Q7: What's the biology behind the algorithm's design choices?

In Fig. 2, notice that the red trapezoid-shaped selection gate is always drawn with a bias toward the right side of the upper right quadrant. This goes back to the principle discussed in Q4 where higher expression of the mutant GOI (y-axis) artificially causes higher reads of protein performance (x-axis) but doesn't necessarily indicate that an individual protein encoded by the mutant GOI is superior. Therefore, cells that demonstrate medium (rather than high) amounts of GOI expression while also yielding high levels of binding are preferable for selection. This strategy is a way of practically normalizing the selection by GOI expression level.

Returning to Fig. 2, notice that the gate is never drawn below a certain soft threshold on the y-axis. This is because lowering the gate too far leads to deleterious mutations that affect the ability of the cell to express protein in the future.

Q8: What reagents will the user need to supply?

The user will need to supply all reagents related to cell culture. This includes 96-well plates (for housing the cells and separately individual experiments run in parallel), cell culture medium (glucose-based for growth and galactose-based for induction), and sterile pipette tips (to avoid contamination across experiments). Additionally, the user will need to provide all protein reagents specific to their experiments. This would include the target protein of an evolving binder or the substrate of an evolving enzyme but also any fluorescently labeled antibodies needed for selection.

Q9: What controls or metrics exist to ensure that selection is proceeding as expected in real-time?

The user is able to visualize the selection decisions made by the robot at any time (similar visualizations to Fig. 2). Additionally, we will develop a feature in the user interface where the researcher can manually draw what they believe is the optimal selection gate and have the software report back the percent difference of selected cells between the robot's classification versus the user's.

Beyond percent difference, the fold-change in the selection pressure used between subsequent cycles (refer to Q4) is a direct measurement of the success of the last selection. This value will be reported to the user to indicate evolutionary progress.

Q10: Is the selection algorithm capable of taking in user feedback in real time?

This feature is not currently available but reinforcement learning approaches may be pursued and integrated in future generations. However, the algorithm is capable of taking in a user input to indicate selection aggressiveness (as discussed in Q4), which is a parameter that is commonly optimized empirically in current-day DE experiments.

Q11: How many directed evolution datasets was this algorithm trained on?

The data used to generate the training and test sets for the selection algorithm were derived from two unique sets of DE experiments conducted in the Chilkoti Lab at Duke University. Each of these DE experiments included three rounds of DE cycles. For these six subsets of data (two experiments x three cycles = six), one million cells were randomly sampled and their 2-dimensional fluorescent readings were used to fit gaussian mixture models. Random noise was added to these models to generate new sets of data so that each cycle number (1-3) had 22 datasets of one million cells each (further details included in technical details).

In more ideal circumstances, we would have trained the algorithm purely on experimental data rather than the gaussian mixture model approach described above; however, we were only able to gain access to two unique DE experiments, which is an inadequate amount of data to train an algorithm.

Q12: What if contamination occurs?

The robot is capable of detecting contamination in two ways: overgrowth and undergrowth. In practice, if selected cells do not recover to an OD of 0.8 within 72 hours of glucose culturing, contamination is suspected and the cells are likely dead. Conversely, if cells grow at a hyper accelerated rate (reaching an OD of 0.8 in less than 24 hours), bacterial or fungal contamination is suspected. In both these cases, if expected OD measurements do not occur within the correct time windows, the robot will notify the user and terminate the experiment.

Technical Details / Preliminary Results

Data Generation

Scatterplots of DE data across different cycles can be difficult to parametrize and model because of the high variability across datasets of different proteins and an incomplete understanding of all the biological factors that influence the speed and quality of a given protein's evolution. Therefore, our approach to generating more data was to fit Gaussian Mixture Models (GMM) to the two sets of DE data (each with three cycles of evolution, which totals six GMMs) and apply empirically determined noise to the covariances of the individuals gaussian distributions (see Figure 3).

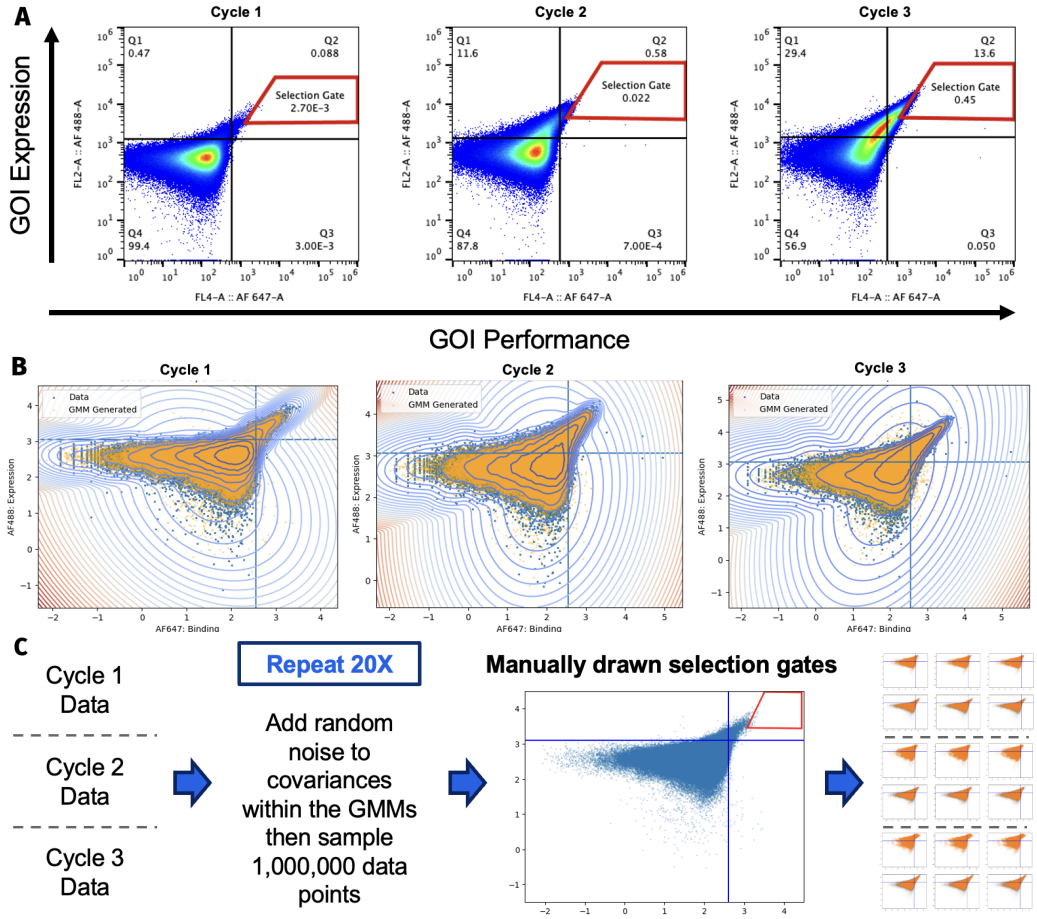


Figure 3: Modeling directed evolution data. A) Flow cytometry plots (GOI expression v. GOI performance) on an evolving population of cells through three DE cycles. B) Fitting GMMs to each DE cycle's data and plotting their levels sets and 1,000,000 samples. C) Workflow of data preparation: for each cycle, a GMM was fitted to the true DE data, noise was introduced to produce 20 unique sets of data/cycle, and a domain expert manually drew selection gates to label the data.

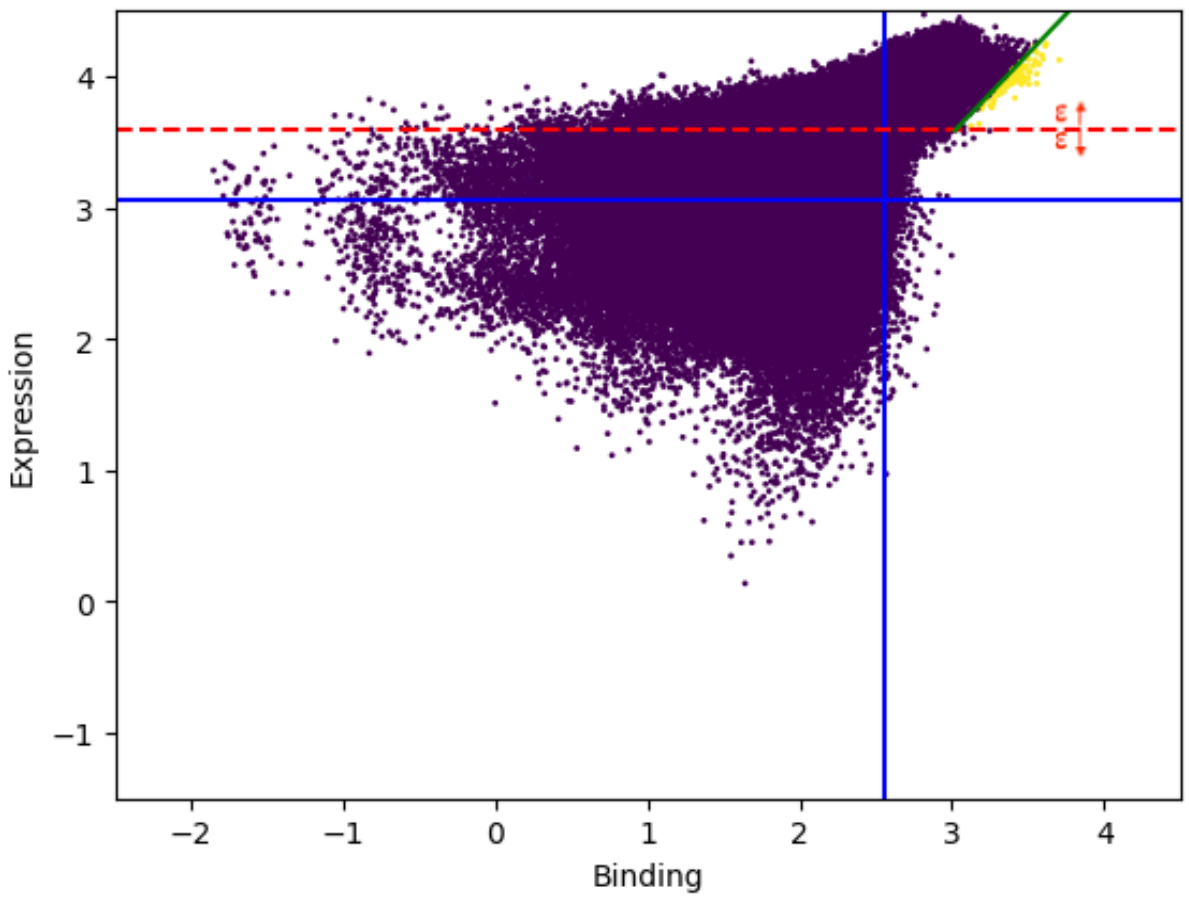
For each cycle (1-3), 20 sets of data were generated (10 from each of the two DE experiments). Including the two original DE datasets used to generate the GMMs, this totaled 22 DE datasets to construct, train, and test our selection algorithm.

Each data frame has data of the form $\{(\boldsymbol{X}_i, Y_i)\}_{i=1}^{n} : \boldsymbol{X}_i = \begin{pmatrix} \text{AF 488-A (Expression)} \\ \text{AF 647-A (Binding)} \end{pmatrix}^T,\ Y_i = \mathbf{1}_{\text{Selected}}$

Algorithm Design:

We implement a streaming support vector machine trained on the sets from cycles 1 and 2 using stochastic gradient descent, then validated and tested on each half of the cycle 3 data (we tried training on data from all three cycles, but this resulted in extremely small quantities of predicted selections). At first glance, this may appear to be a problem that warrants a nonlinear SVM. However, notice in the figure below that the axes aren't in their usual places. The x and y axes here represent minimum permissible values for binding and expression, respectively. We noticed that the selection boundary can always be drawn such that it has a line parallel to the x-axis. After consulting with our resident domain expert, we deemed it appropriate to set a threshold for expression above the established one, allowing us to fit a standard linear SVM. We naively set this threshold at the minimal observed expression value over all selected cells from the training data, and allow for a tunable ε-window around it in the model. For a clearer picture of what's going on, see the image below. The best cells are those with the highest degrees of binding who also have the highest ratios of binding to expression. What we aim to do is find the ideal shift of the red dashed line so that we can consistently draw the green line that optimally separates the best cells from the rest.

Because this is a rather subjective process in practice (different scientists are unlikely to select the exact same set of cells), loss is difficult to conceptualize. We use hinge loss to train our SVM, but focus solely on precision over the validation and test sets, as we want to minimize false positives.



The algorithm proceeds as follows:

1. Iterate over a range of α and ε values to find the optimal choices
    i. Initialize the SGD classifier with hinge loss
    ii. For each data sample in the training data:
        a. Select only the points above the threshold ($\pm\varepsilon$)
        b. Standardize the data
        c. Oversample the response (the data is very imbalanced; ~100 selected cells per 1,000,000)
        d. Update the streaming estimator
    iii. Repeat (ii.) over the validation set and store the best precision, along with the corresponding $\alpha$and ε
2. Fit the model once more with the discovered optimal hyperparameters

Results and Potential Areas for Improvement

We obtained a maximum precision score of 0.812 with α = 1e-6 and ε = 0.1 on our validation set. Note that we had a higher precision when training on data from all cycles,

but again this was due to the extremely small selection size (so it would have had a much lower F1 score, for instance).

Given the rather simple selection criteria, we believe it may be possible to implement a much simpler model that could yield similarly good (if not better) results. Consider for instance the following:

1. Given our threshold y=*b*, choose the top *n* (by binding value) remaining cells above *b* (or the top *p*%)
2. Partition the range of these binding values into subsets c_k, then select the *m* cells over each subset with the highest binding:expression ratios, such that *mk* equals the desired number (or such that *mk/N*, [where *N* is the total number of cells] equals the desired proportion) of selected cells

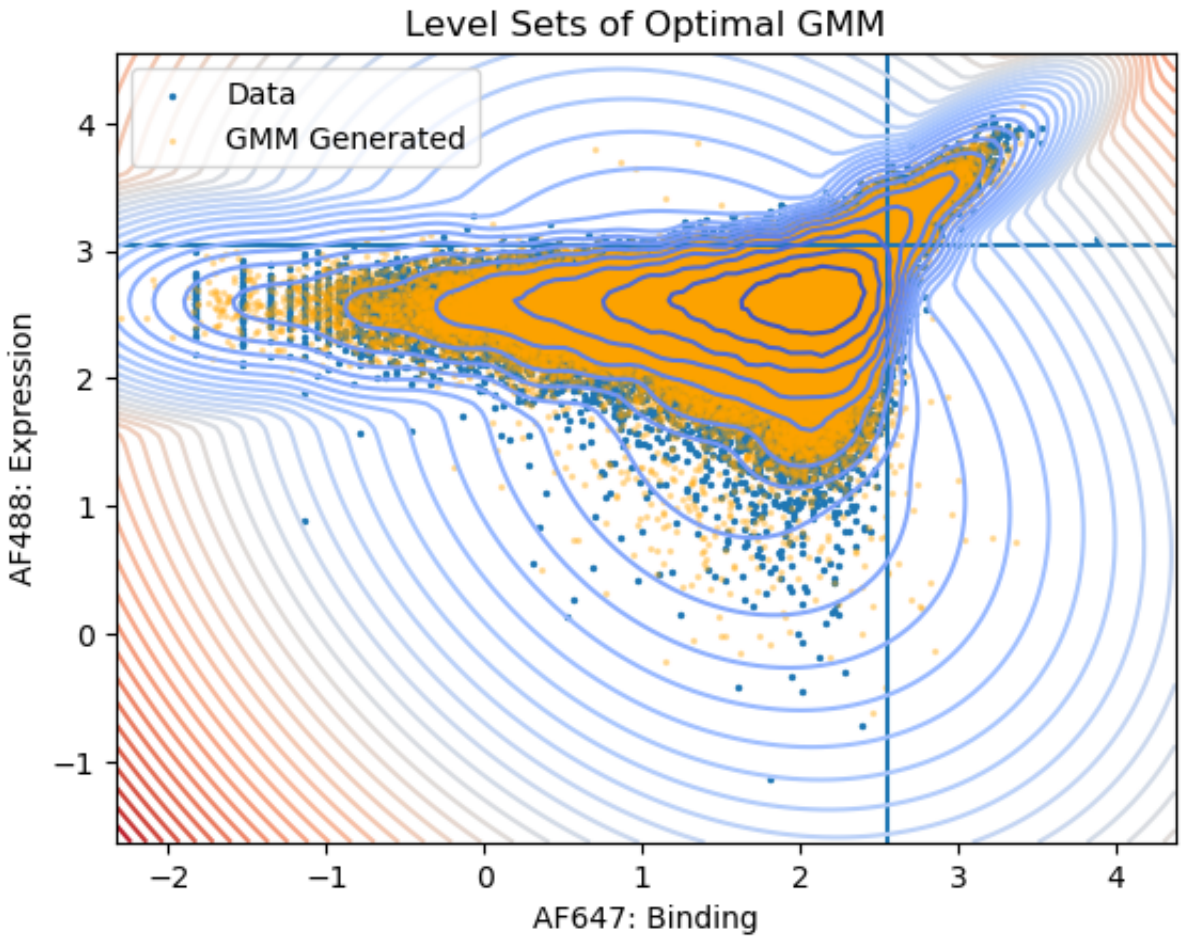
This was unfortunately an idea that we did not arrive at until rather late in the project, and hasty implementations of it yielded worse results than the SVM.

Data Generation

In [241]:
```
def createGMM(data):
    ldata = np.log10(data[(data>0).all(axis=1)])
    gmm = GaussianMixture(n_components=75, covariance_type='full')
    gmm.fit(ldata)

    samples, _ = gmm.sample(1000000)

    plt.scatter(ldata['AF 647-A'],ldata['AF 488-A'], s=2, label='Data'
    plt.scatter(samples[:,1], samples[:,0], color= 'orange', alpha = 0

    plt.axhline(y=3.05)
    plt.axvline(x=2.55)

    # Plot the level sets of each Gaussian component
    x = np.linspace(np.min(ldata['AF 488-A'])-0.5, np.max(ldata['AF 48
    y = np.linspace(np.min(ldata['AF 647-A'])-0.5, np.max(ldata['AF 64
    X, Y = np.meshgrid(x, y)
    XX = np.array([X.ravel(), Y.ravel()]).T
    Z = -gmm.score_samples(XX)
    Z = Z.reshape(X.shape)

    plt.contour(Y, X, Z, levels=np.linspace(Z.min(), Z.max(), 40), cma

    plt.ylabel('AF488: Expression')
    plt.xlabel('AF647: Binding')
    plt.title('Level Sets of Optimal GMM')
    plt.legend()
    plt.show()
    return ldata, gmm
```

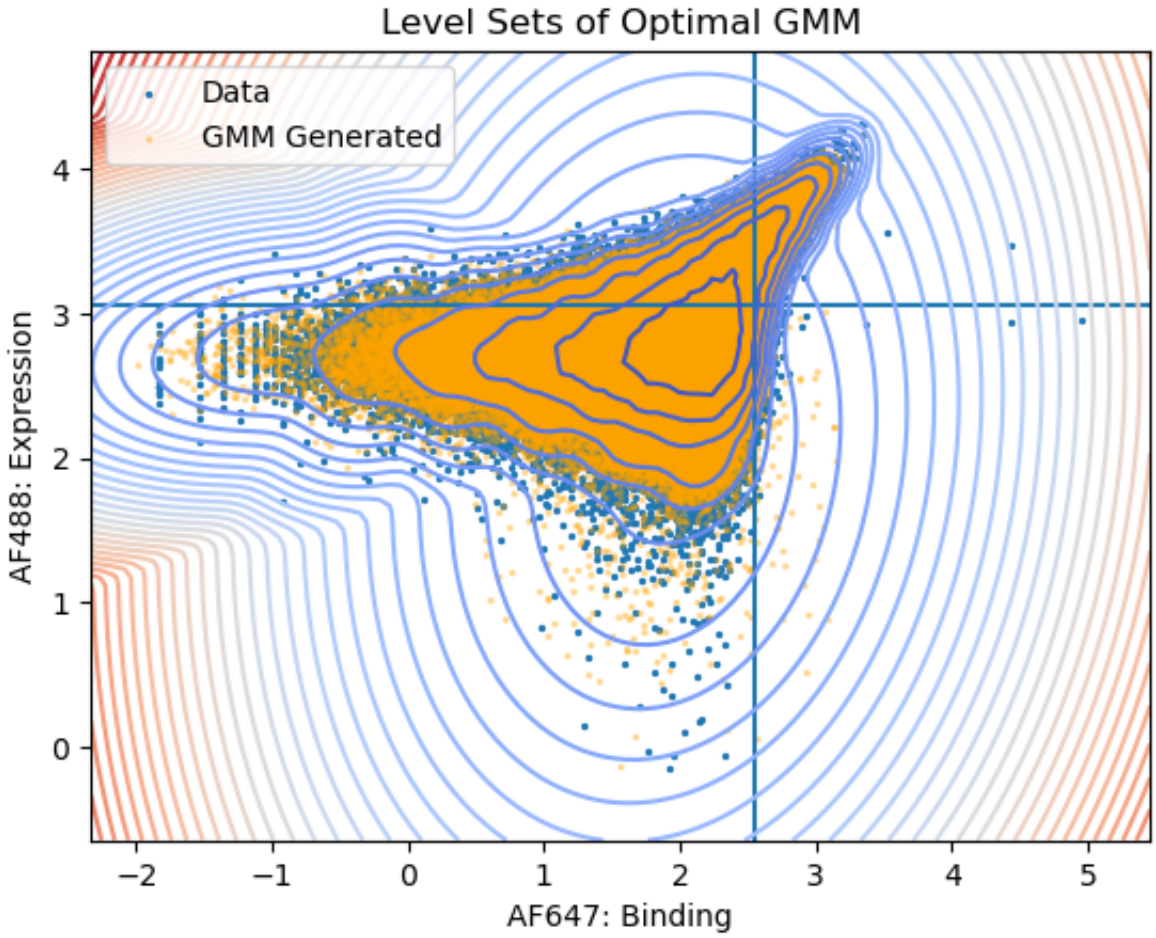
In [242]:
```
%matplotlib inline
ldF1data, ldF1gmm = createGMM(dsGP_F1)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```



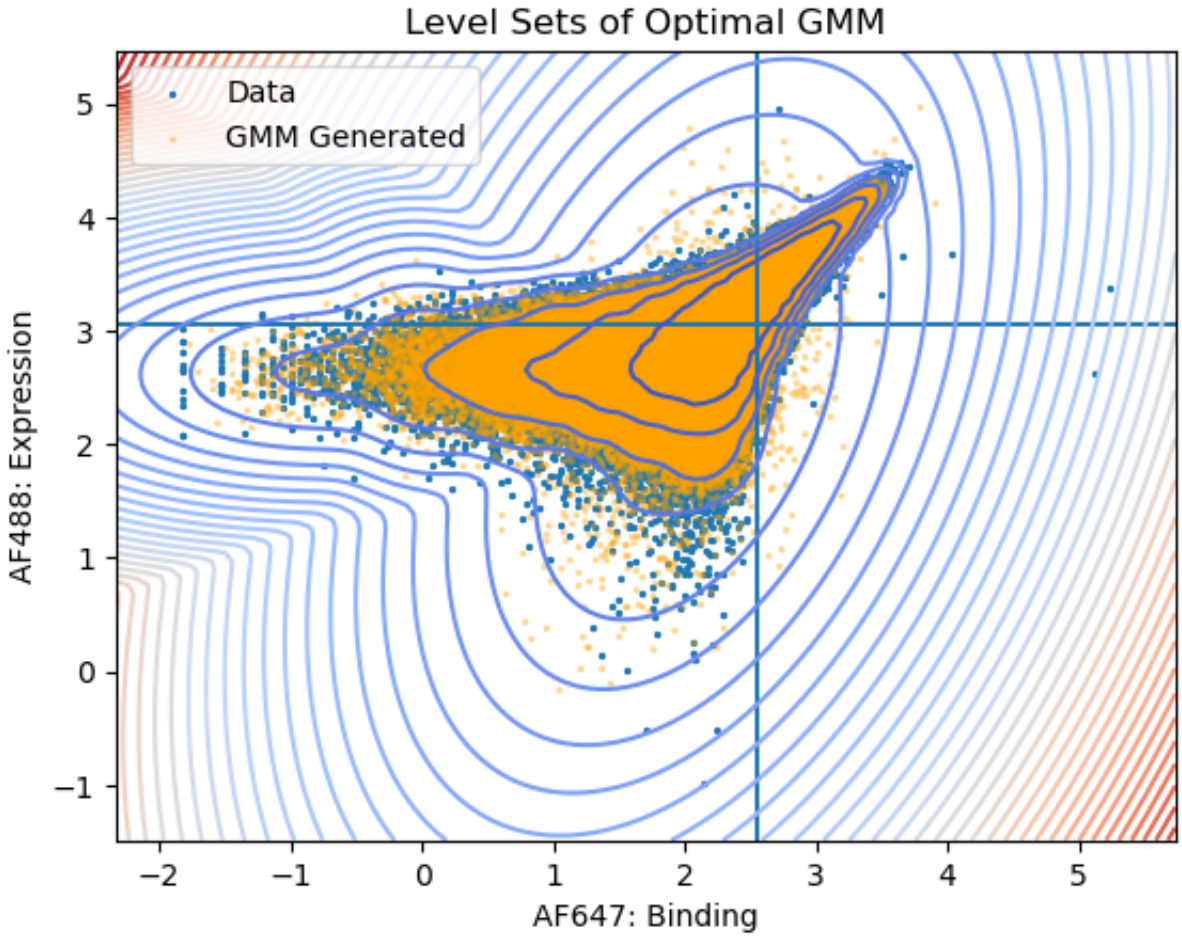
In [243]:
```
ldF2data, ldF2gmm = createGMM(dsGP_F2)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```



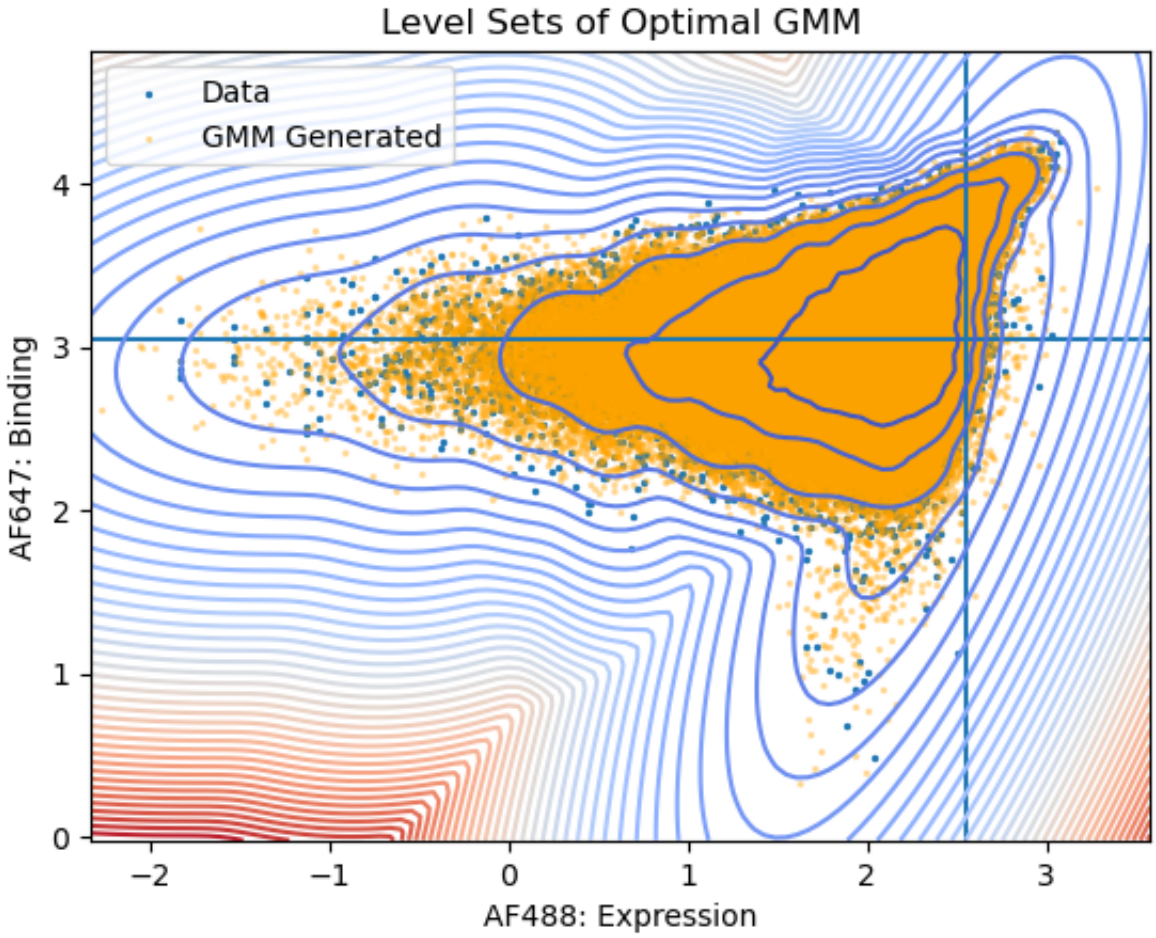
In [244]:
```
ldF3data, ldF3gmm = createGMM(dsGP_F3)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```



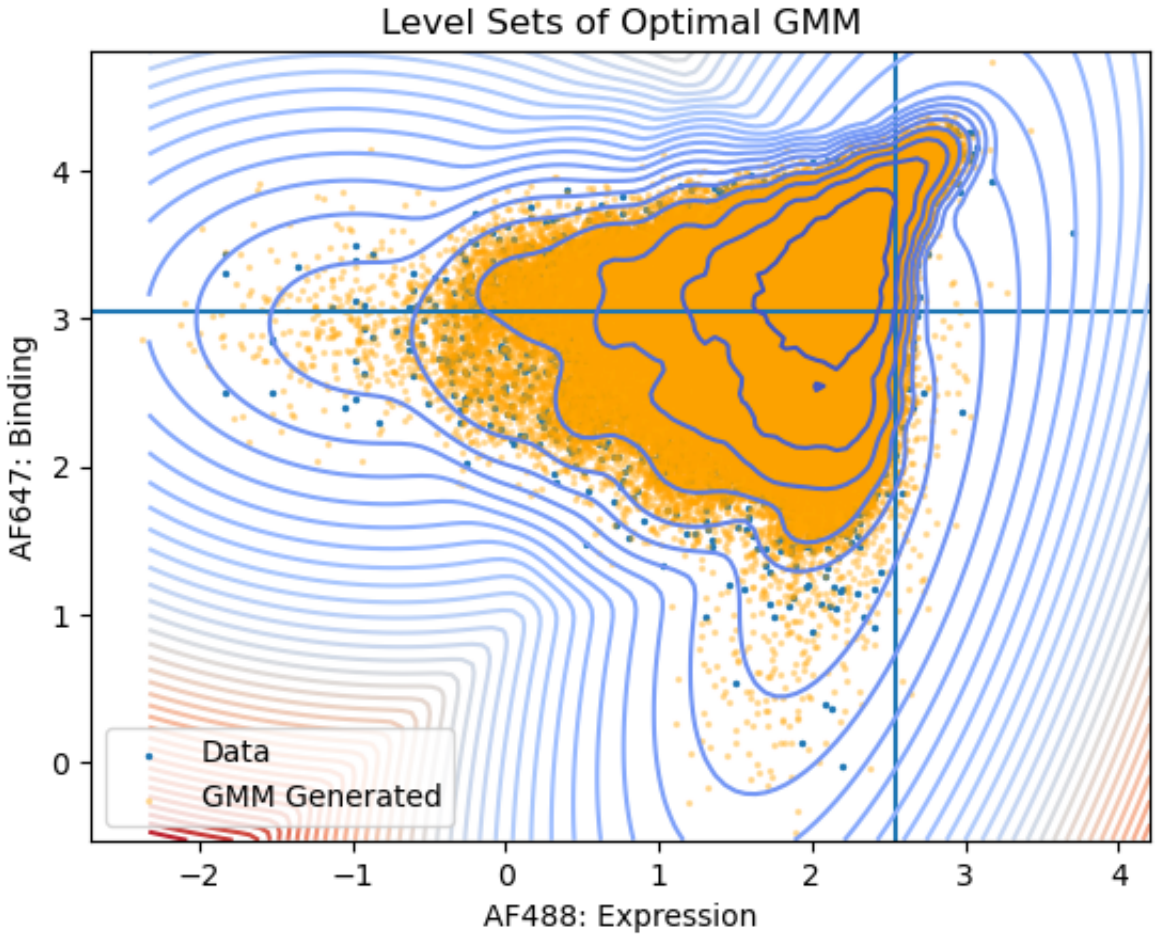
In [214]:
```
lmF1data, lmF1gmm = createGMM(msGP_F1)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```



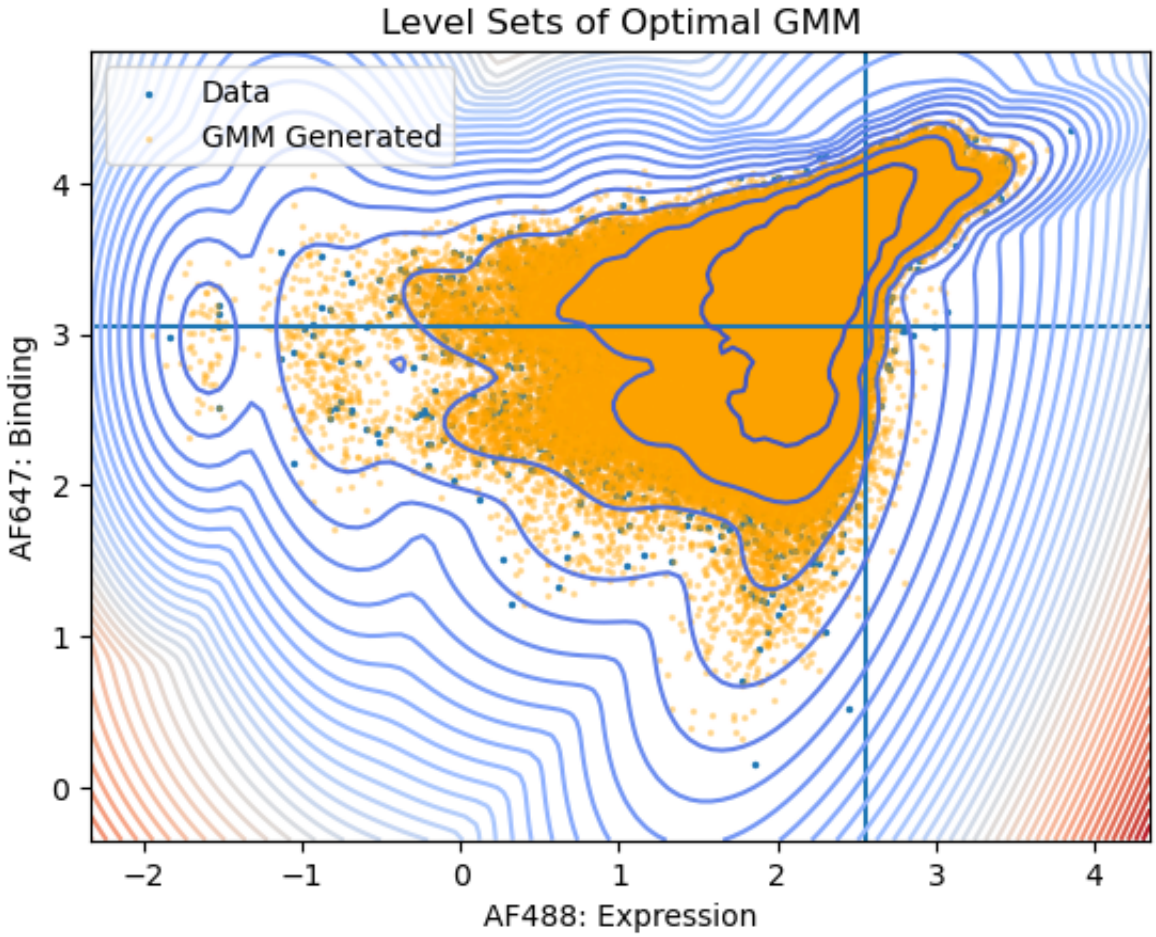
In [215]:
```
lmF2data, lmF2gmm = createGMM(msGP_F2)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```



In [216]:
```
lmF3data, lmF3gmm = createGMM(msGP_F3)
```

```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/base.py:409: UserWarning: X does not have valid feature names, but GaussianMixture was fitted with feature names
  warnings.warn(
```

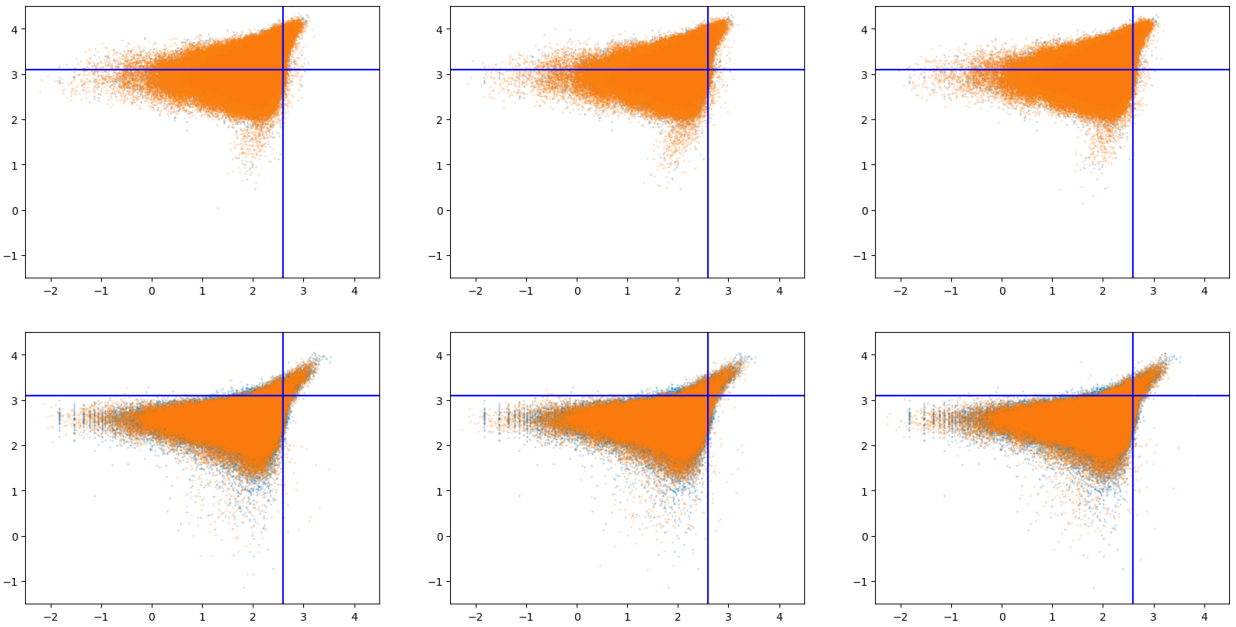


In [218]:
```
def makevar(data_m, data_d, gmm_m, gmm_d, n):
    cstdev = 0.0001
    #mstdev = 0.0005
    #mmstdev = 0.0001

    #mean_mo = np.copy(gmm_m.means_)
    cov_mo = np.copy(gmm_m.covariances_)
    #mean_do = np.copy(gmm_d.means_)
    cov_do = np.copy(gmm_d.covariances_)
    t1 = 3.05
    t2 = 2.55
    store = np.empty(n+2, dtype=object)
    fig, axn = plt.subplots(2,3,figsize = (20,10))
    for ax in axn.flat:
        ax.set_xlim([-2.5, 4.5])
```

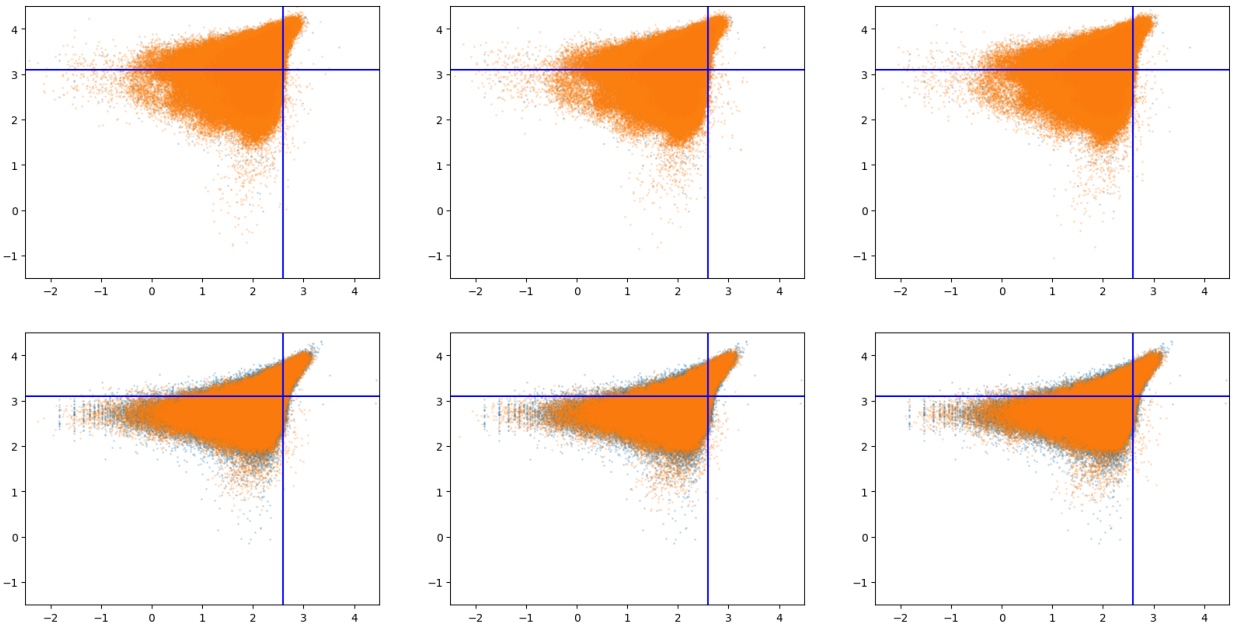
```
        ax.set_ylim([-1.5, 4.5])
        ax.axhline(y=3.1, c = 'b')
        ax.axvline(x=2.6, c = 'b')
    stdevs = np.linspace(0.01, 0.11, 20)
    for i in range(int(n/2)):
        #sample_m = gmm_m.sample(1000000)[0]
        #gmm_m.means_ = mean_mo + np.random.normal(loc=0, scale=mmstde
        gmm_m.covariances_ = cov_mo + np.random.normal(loc=0, scale=[[
        #mask_m = (sample_m[:, 0] > t1) & (sample_m[:, 1] > t2)
        #noise_m = np.random.normal(loc=0, scale=stdevs[i], size = sam
        #sample_m[mask_m] = sample_m[mask_m] + noise_m[mask_m]
        store[i*2] = gmm_m.sample(1000000)[0]
        #gmm_m.covariances_ = cov_mo

        #gmm_d.means_ = mean_do + np.random.normal(loc=0, scale=mstdev
        gmm_d.covariances_ = cov_do + np.random.normal(loc=0, scale=[[
        #sample_d = gmm_d.sample(1000000)[0]
        #mask_d = (sample_d[:, 0] > t1) & (sample_d[:, 1] > t2)
        #noise_d = np.random.normal(loc=0, scale=stdevs[i], size = sam
        #sample_d[mask_d] = sample_d[mask_d] + noise_d[mask_d]
        store[i*2+1] = gmm_d.sample(1000000)[0]
        #gmm_d.covariances_ = cov_do

        if i < 3:
            axn[0,i].scatter(data_m['AF 647-A'], data_m['AF 488-A'], s
            axn[0,i].scatter(store[i*2][:,1], store[i*2][:,0],s=1, alp

            axn[1,i].scatter(data_d['AF 647-A'], data_d['AF 488-A'], s
            axn[1,i].scatter(store[i*2+1][:,1], store[i*2+1][:,0],s=1,

    #gmm_m.means_ = mean_mo
    gmm_m.covariances_ = cov_mo
    #gmm_d.means_ = mean_do
    gmm_d.covariances_ = cov_do
    store[n] = data_m
    store[n+1] = data_d
    return store
```

In [219]:
```
F1data = makevar(lmF1data, ldF1data,lmF1gmm,ldF1gmm, 20)
```

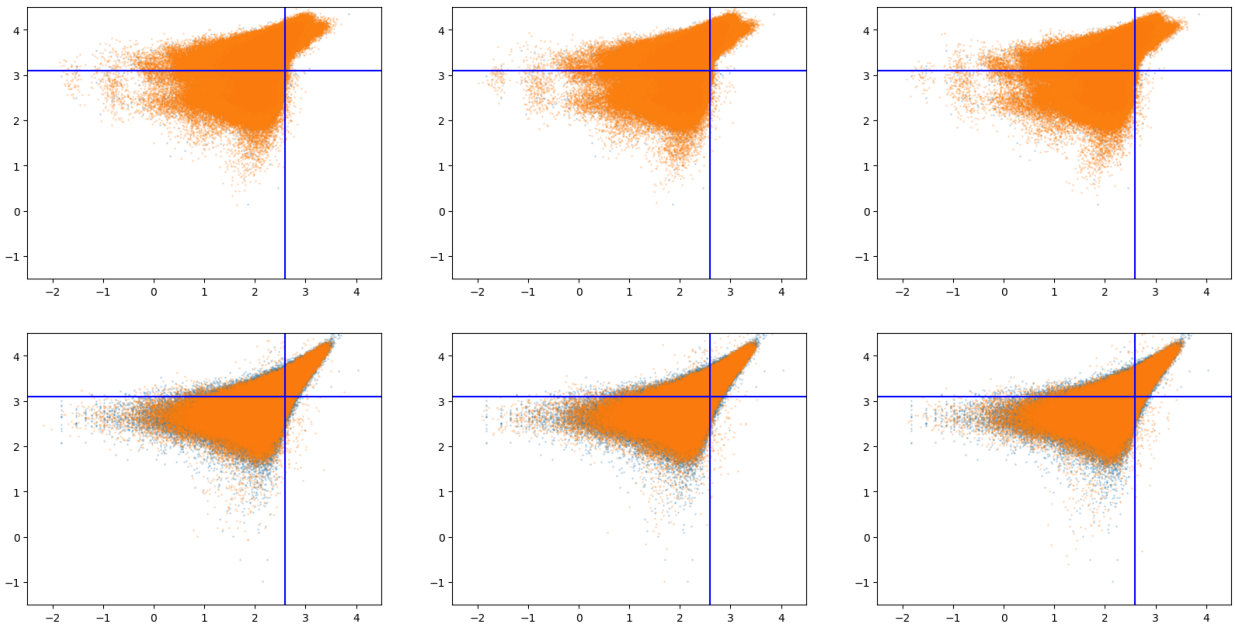
```
/opt/anaconda3/envs/compsci527/lib/python3.10/site-packages/sklearn/mixture/_base.py:440: RuntimeWarning: covariance is not positive-semidefinite.
  rng.multivariate_normal(mean, covariance, int(sample))
```



In [220]:
```
F2data = makevar(lmF2data, ldF2data,lmF2gmm,ldF2gmm, 20)
```



In [221]:
```
F3data = makevar(lmF3data, ldF3data,lmF3gmm,ldF3gmm, 20)
```



In [224]:
```
def select(data):
    %matplotlib qt
    labeled = np.empty(len(data), dtype = object)
    for i in range(len(data)):
        dataset = pd.DataFrame(data[i], columns = ['AF 488-A', 'AF 647

        plt.scatter(dataset['AF 647-A'], dataset['AF 488-A'], s = 1, a
        plt.xlim([-2.5, 4.5])
        plt.ylim([-1.5, 4.5])
        plt.axhline(y=3.1)
        plt.axvline(x=2.6)

        plt.ion()
        gate_co = plt.ginput(n=4, timeout = 0)
        plt.close()
        plt.ioff()

        gate_path = Path(gate_co + [gate_co[0]])
        inside_gate = gate_path.contains_points(dataset[['AF 647-A', '
        selected = inside_gate.astype(int)
        dataset['selected'] = selected
        labeled[i] = dataset.copy()
    return labeled
```

In [225]:
```
F1labeled = select(F1data)
```

In [232]:
```
F2labeled = select(F2data)
```

In [234]:
```
F3labeled = select(F3data)
```

In [228]:
```
import pickle
```

In [229]:
```
with open('/Users/joannapeng/Desktop/ECE 682/Project/F1labeled.pkl', '
    pickle.dump(F1labeled, file)
```

In [233]:
```
with open('/Users/joannapeng/Desktop/ECE 682/Project/F2labeled.pkl', '
    pickle.dump(F2labeled, file)
```

In [235]:
```
with open('/Users/joannapeng/Desktop/ECE 682/Project/F3labeled.pkl', '
    pickle.dump(F3labeled, file)
```

In [ ]:
```

```

In [230]:
```
with open('/Users/joannapeng/Desktop/ECE 682/Project/F1labeled.pkl', '
    loaded_dataframes = pickle.load(file)
```

In [231]:
```
loaded_dataframes[0]
```

Out[231]:

	AF 488-A	AF 647-A	selected
0	2.533940	1.898733	0
1	2.580516	2.107764	0
2	2.538185	1.872136	0
3	2.476577	1.780025	0
4	2.636533	1.839921	0
...	...	...	...
999995	2.656157	0.199647	0
999996	2.827302	0.082376	0
999997	3.056665	-0.016329	0
999998	2.645850	0.188700	0
999999	2.784102	0.645337	0

1000000 rows × 3 columns

Model Generation and Results

In [1]:
```
import numpy as np
import pandas as pd
import math
import matplotlib
import matplotlib.pyplot as plt
import scipy as sci
from sklearn.mixture import GaussianMixture
from matplotlib.path import Path
import pickle
```

Open the three sets of data. Each pkl file represents a specific round of evolution and holds a set of 22 dataframes. F1 means those DataFrames capture data of cells that haven't gone through selection yet. F2 means these cells have gone through one round of selection and are on their second round of selection.

In [2]:
```
with open('F1labeled.pkl', 'rb') as file:
    F1labeled = pd.read_pickle(file)
with open('F2labeled.pkl', 'rb') as file:
    F2labeled = pd.read_pickle(file)
with open('F3labeled.pkl', 'rb') as file:
    F3labeled = pd.read_pickle(file)
```

Each pickle file is a list of pandas DataFrames. Within the DataFrames are three columns. The first two columns specify the cell's "coordinates." More specifically, 'AF 488-A' measures the amount of protein expression on the cell surface (this should be plotted on the y axis, just common practice) and 'AF 647-A' measures the protein's functionality/performance (this should be plotted on the x axis). The last column of the DataFrame indicates whether a cell was selected (1) or not (0) to continue on to the next round of evolution

In [3]:
```
F1labeled[0].head()
```

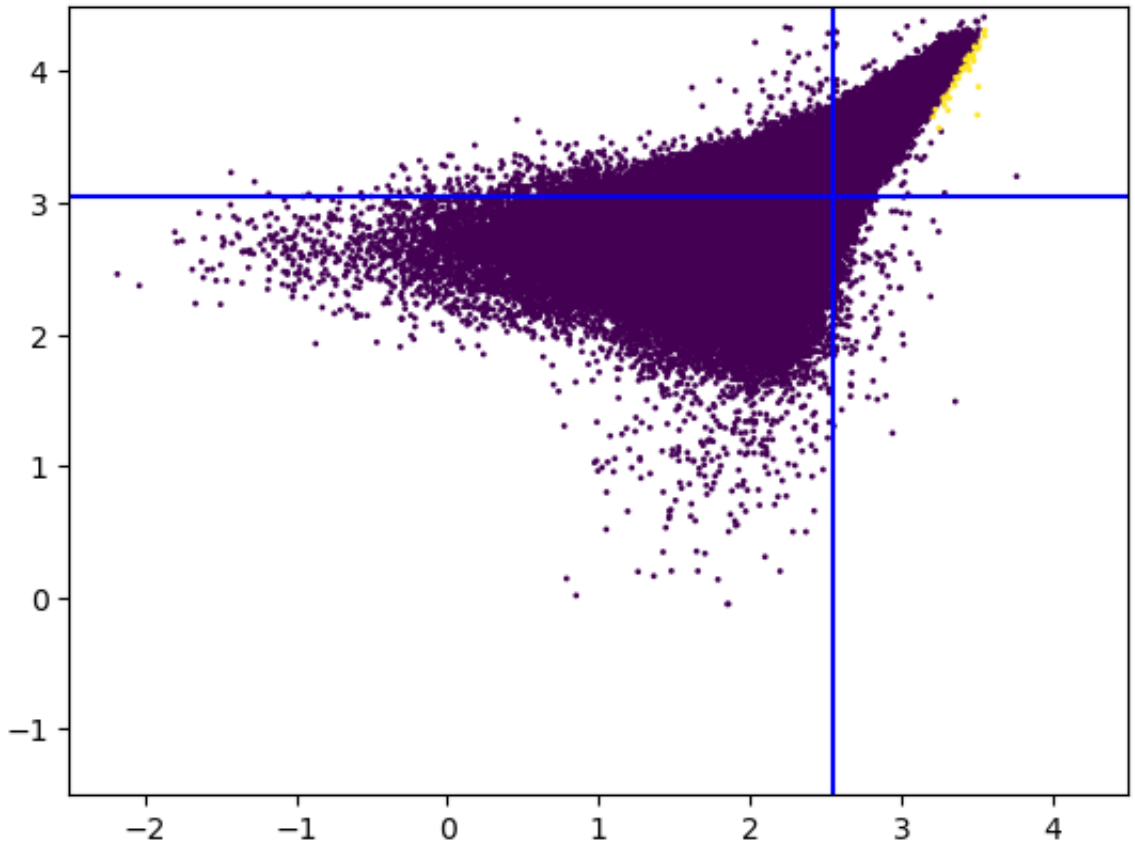
Out[3]:

	AF 488-A	AF 647-A	selected
0	2.533940	1.898733	0
1	2.580516	2.107764	0
2	2.538185	1.872136	0
3	2.476577	1.780025	0
4	2.636533	1.839921	0

Example for visualizing data. I would recommend plotting at a couple different indices so that you can get an idea of the diversity in each pkl file and to see the slight differences that acrue over F1-F2-F3

In [4]:
```
plt.scatter(F3labeled[1]['AF 647-A'], F3labeled[1]['AF 488-A'], s = 1,
plt.xlim([-2.5, 4.5])
plt.ylim([-1.5, 4.5])
# plt.xlabel("Binding")
# plt.ylabel("Expression")
plt.axhline(y=3.05, c = 'b')
plt.axvline(x=2.55, c = 'b')
```

Out[4]: <matplotlib.lines.Line2D at 0x177ad1e50>



Selecting the Threshold

The threshold will be the minimum Expression value for selected cells across all labeled data.

In [5]:
```
# arbitrarily large threshold
threshold = 1e6

for frames in [F1labeled, F2labeled, F3labeled]:
    for df in frames:
        selected_df = df[df['selected'] == 1]

        # check if minimum value selected is lower than threshold
        min_value = selected_df['AF 488-A'].min()
        if min_value < threshold:
            threshold = min_value

print(threshold)
```

```
3.4772984686494017
```

Training the Model:

In [6]:
```
def Xy_preprocess(frame, threshold):
    from sklearn.preprocessing import StandardScaler, LabelEncoder

    scaler = StandardScaler()

    # ensure data is over threshold
    frame = frame[frame['AF 488-A'] > threshold]
    X = frame[['AF 488-A', 'AF 647-A']]

    # standardize data
    X = scaler.fit_transform(X)

    # encode response
    label_encoder = LabelEncoder()
    y = label_encoder.fit_transform(frame['selected'])

    return X, y
```

In [9]:
```
import warnings
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score, precision_score, f1_score
from imblearn.over_sampling import SMOTE

warnings.filterwarnings('ignore')
```

```
validation_range = np.arange(0, 11)
test_range = np.arange(11, 22)

smote = SMOTE(random_state=29)

# ranges are small, bc I tested larger ones, and now it's just to show
alphas = [.000001, .00005, .00001]
epsilons = [-.1, -.01, 0, .01, .1]

best_score = 0
for a in alphas:
    for e in epsilons:
        # hinge loss necessary to implement SVM (per documentation)
        svm = SGDClassifier(loss='hinge', alpha=a, random_state=29)

        # fit model on training
        for frames in [F1labeled, F2labeled]:
            for frame in frames:
                new_thresh = threshold + e
                X, y = Xy_preprocess(frame, new_thresh)

                # oversample response so that prediction can be better
                X_resampled, y_resampled = smote.fit_resample(X, y)

                # update streaming estimator
                svm.partial_fit(X_resampled, y_resampled, classes = np

        mean_precision_scores = []

        # loop over validation set to evaluate
        for i in validation_range:
            X, y = Xy_preprocess(F3labeled[i], new_thresh)
            if svm.predict(X).sum() == 0:
                print('broken!')
            # focus on precision, as we want highest proportion of tru
            mean_precision = np.mean(cross_val_score(svm, X, y, scorin

            mean_precision_scores.append(mean_precision)

        if np.mean(mean_precision_scores) > best_score:
            best_score = np.mean(mean_precision_scores)
            best_alpha = a
            best_epsilon = e

print(f'Best Precision Score: {best_score}')
print(f'Alpha: {best_alpha}')
print(f'Epsilon: {best_epsilon}')
```

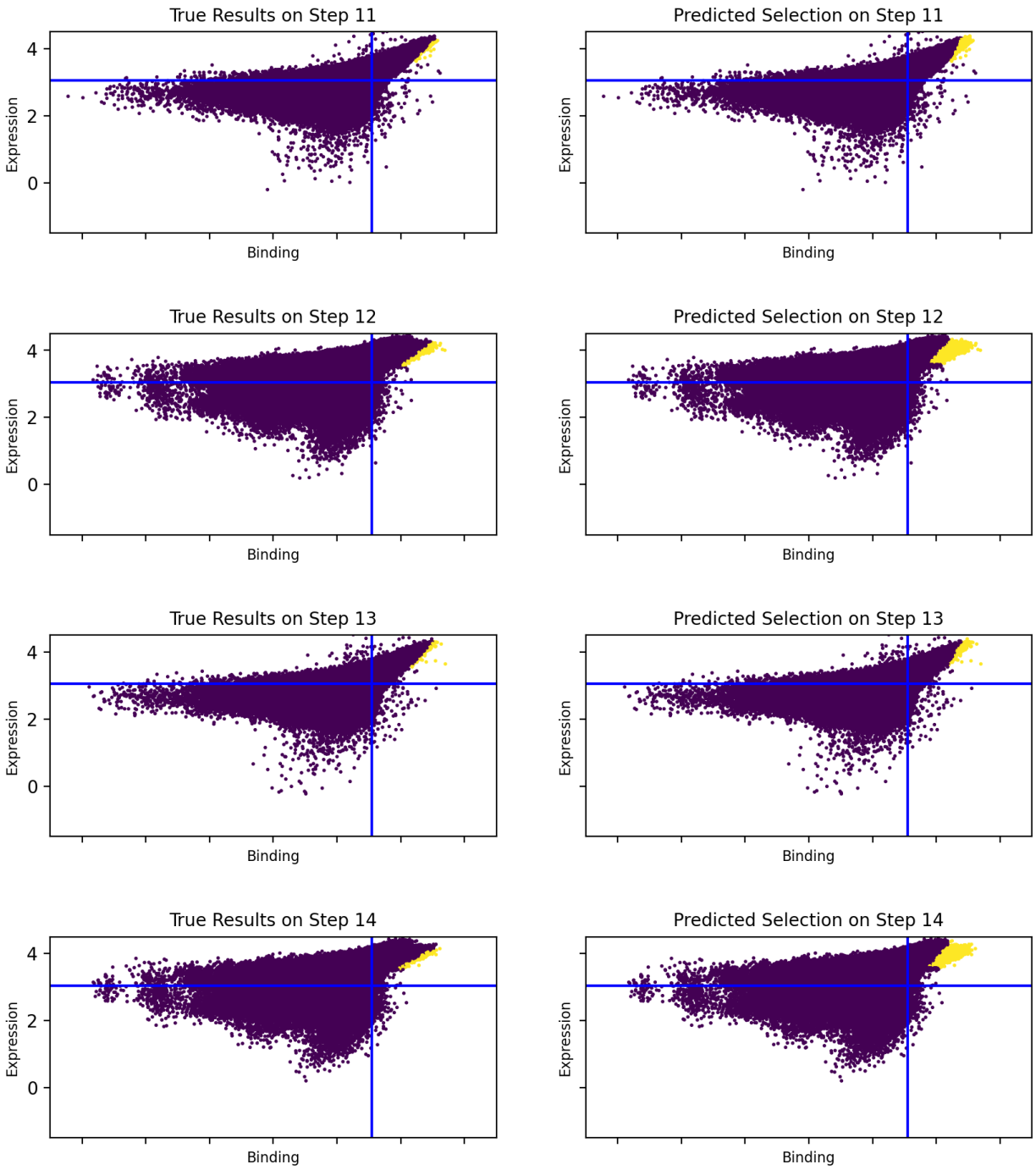
```
Best Precision Score: 0.8120798619119067
Alpha: 1e-06
```

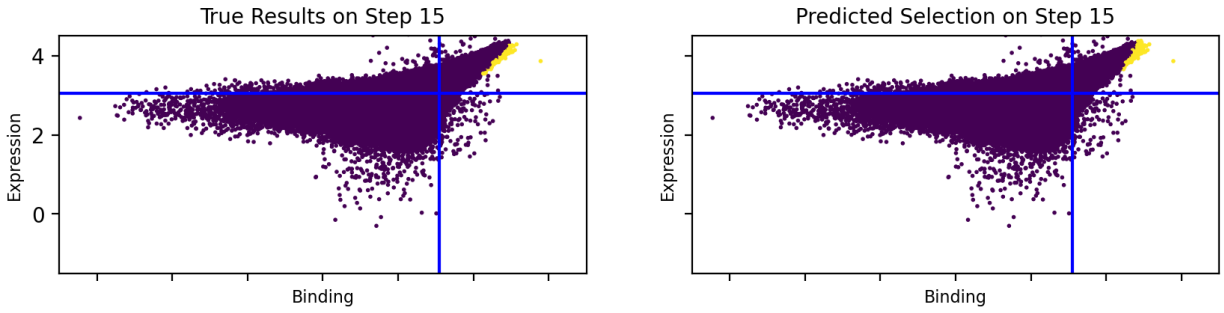
```
Epsilon: 0.1
```

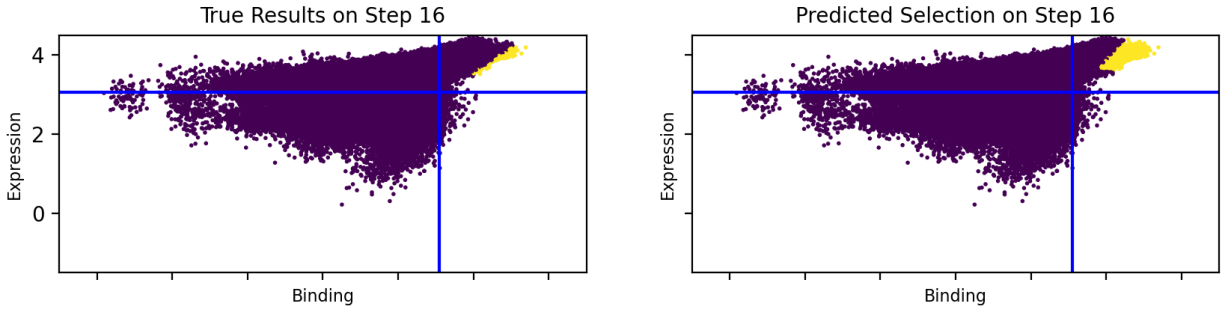
In [10]:
```
# replicate process to fit the model once the hyperparameters are foun
smote = SMOTE(random_state=29)

svm = SGDClassifier(loss='hinge', alpha=best_alpha, random_state=29)
threshold = threshold + best_epsilon

# fit model on training
for frames in [F1labeled, F2labeled]:
    for frame in frames:
        X, y = Xy_preprocess(frame, threshold)

        # oversample response so that prediction can be better
        X_resampled, y_resampled = smote.fit_resample(X, y)

        # update streaming estimator
        svm.partial_fit(X_resampled, y_resampled, classes = np.unique(
```

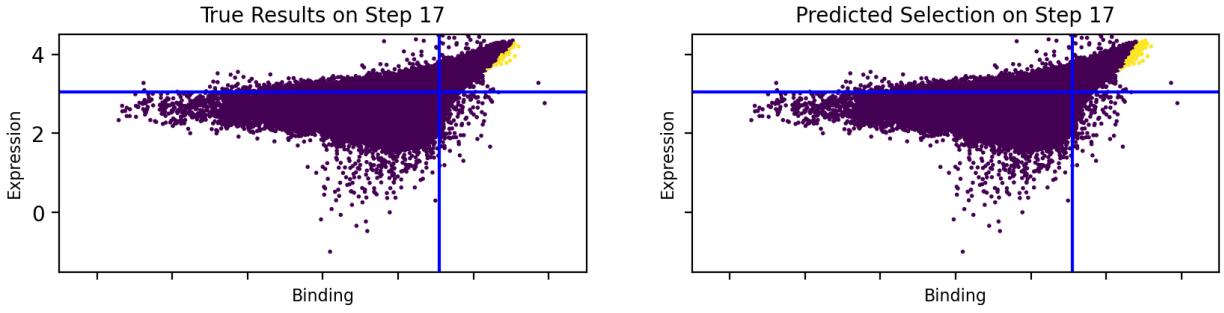
In [11]:
```
fig, axs = plt.subplots(11, 2, figsize = (10, 33), sharey = True, shar

for i in range(11):
    # narrow range to test set
    X_new = F3labeled[11+i][F3labeled[11+i]['AF 488-A'] > threshold][[

    # scale data
    scaler = StandardScaler()
    X_new = scaler.fit_transform(X_new)

    # predict response
    predicted = svm.predict(X_new)

    # identify un-included data
    not_included = F3labeled[11+i][F3labeled[11+i]['AF 488-A'] <= thre
    not_included['predicted'] = 0

    # return results to regular scale
    results = pd.DataFrame({"AF 488-A": X_new[:,0], "AF 647-A": X_new[
    results[['AF 488-A', 'AF 647-A']] = scaler.inverse_transform(resul

    # combine un-included and included data
    results = pd.concat([results, not_included], axis = 0, ignore_inde

    # visualize data
    axs[i,0].scatter(F3labeled[11+i]['AF 647-A'], F3labeled[11+i]['AF
    axs[i,1].scatter(results['AF 647-A'], results['AF 488-A'], s=1, c

    for j in range(2):
        axs[i,j].set_xlim([-2.5, 4.5])
```

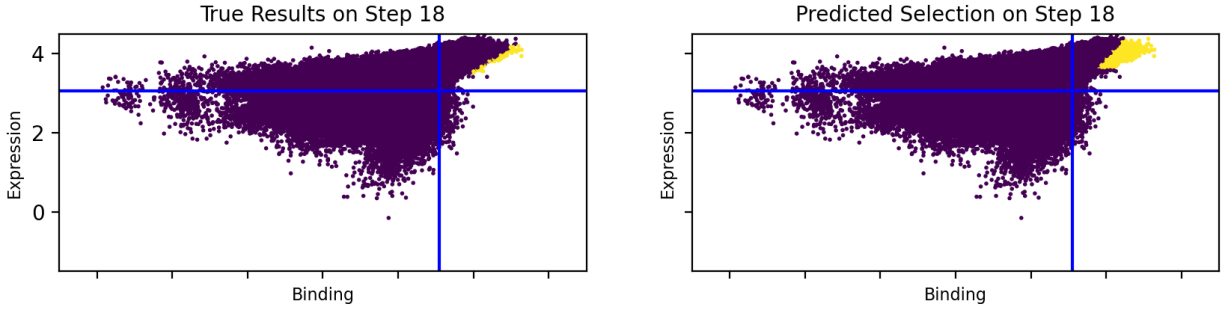
```
        axs[i,j].set_ylim([-1.5, 4.5])
        axs[i,j].set_xlabel("Binding", fontsize = 8)
        axs[i,j].set_ylabel("Expression", fontsize = 8)
        axs[i,j].axhline(y=3.05, c = 'b')
        axs[i,j].axvline(x=2.55, c = 'b')

    axs[i,0].set_title(f"True Results on Step {11+i}", fontsize = 10)
    axs[i,1].set_title(f"Predicted Selection on Step {11+i}", fontsize

plt.subplots_adjust(hspace=0.5)
```

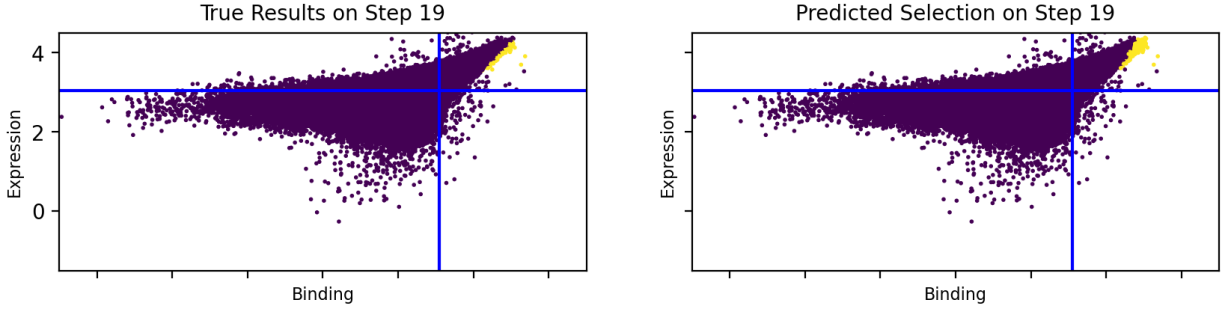



True Results on Step 15
Predicted Selection on Step 15
Expression
Binding
4
2
0

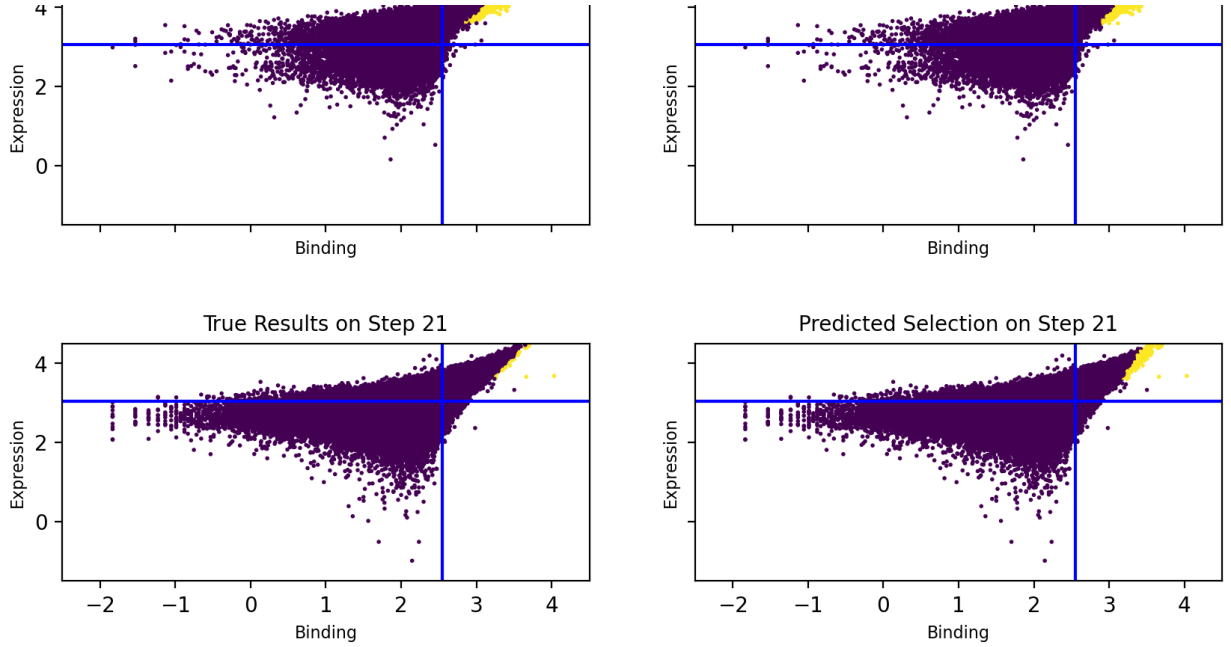

True Results on Step 16
Predicted Selection on Step 16
Expression
Binding
4
2
0


True Results on Step 17
Predicted Selection on Step 17
Expression
Binding
4
2
0


True Results on Step 18
Predicted Selection on Step 18
Expression
Binding
4
2
0


True Results on Step 19
Predicted Selection on Step 19
Expression
Binding
4
2
0


True Results on Step 20
Predicted Selection on Step 20



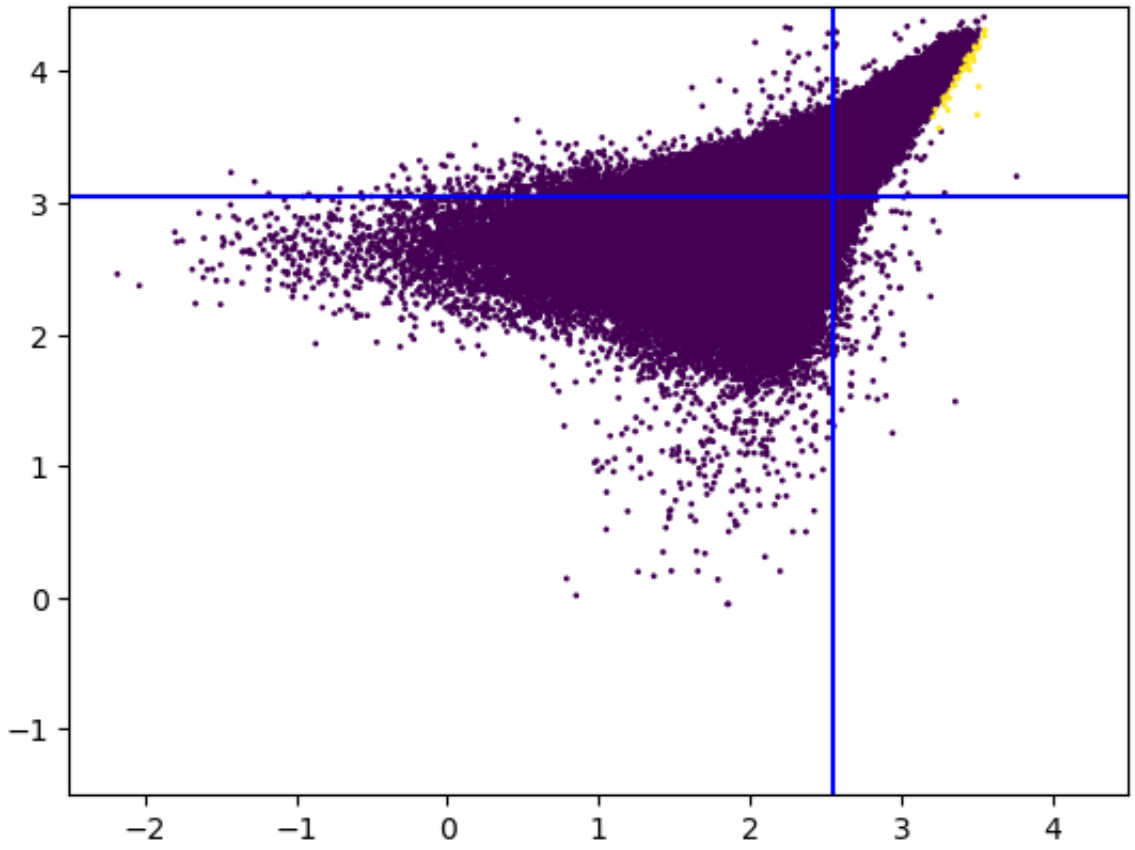
In [ ]:
```
import joblib

fig.savefig('test_results.jpeg')
joblib.dump(svm, 'DirectedEvolutionSVM.pkl')
```

In [ ]:
```

```

Example for visualizing data. I would recommend plotting at a couple different indices so that you can get an idea of the diversity in each pkl file and to see the slight differences that acrue over F1-F2-F3

In [4]:
```
plt.scatter(F3labeled[1]['AF 647-A'], F3labeled[1]['AF 488-A'], s = 1,
plt.xlim([-2.5, 4.5])
plt.ylim([-1.5, 4.5])
# plt.xlabel("Binding")
# plt.ylabel("Expression")
plt.axhline(y=3.05, c = 'b')
plt.axvline(x=2.55, c = 'b')
```

Out[4]: <matplotlib.lines.Line2D at 0x177ad1e50>



Selecting the Threshold

The threshold will be the minimum Expression value for selected cells across all labeled data.

In [5]:
```
# arbitrarily large threshold
threshold = 1e6

for frames in [F1labeled, F2labeled, F3labeled]:
    for df in frames:
        selected_df = df[df['selected'] == 1]

        # check if minimum value selected is lower than threshold
        min_value = selected_df['AF 488-A'].min()
        if min_value < threshold:
            threshold = min_value

print(threshold)
```

```
3.4772984686494017
```

Training the Model:

In [6]:
```
def Xy_preprocess(frame, threshold):
    from sklearn.preprocessing import StandardScaler, LabelEncoder

    scaler = StandardScaler()

    # ensure data is over threshold
    frame = frame[frame['AF 488-A'] > threshold]
    X = frame[['AF 488-A', 'AF 647-A']]

    # standardize data
    X = scaler.fit_transform(X)

    # encode response
    label_encoder = LabelEncoder()
    y = label_encoder.fit_transform(frame['selected'])

    return X, y
```

In [9]:
```
import warnings
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score, precision_score, f1_score
from imblearn.over_sampling import SMOTE

warnings.filterwarnings('ignore')
```

```
validation_range = np.arange(0, 11)
test_range = np.arange(11, 22)

smote = SMOTE(random_state=29)

# ranges are small, bc I tested larger ones, and now it's just to show
alphas = [.000001, .00005, .00001]
epsilons = [-.1, -.01, 0, .01, .1]

best_score = 0
for a in alphas:
    for e in epsilons:
        # hinge loss necessary to implement SVM (per documentation)
        svm = SGDClassifier(loss='hinge', alpha=a, random_state=29)

        # fit model on training
        for frames in [F1labeled, F2labeled]:
            for frame in frames:
                new_thresh = threshold + e
                X, y = Xy_preprocess(frame, new_thresh)

                # oversample response so that prediction can be better
                X_resampled, y_resampled = smote.fit_resample(X, y)

                # update streaming estimator
                svm.partial_fit(X_resampled, y_resampled, classes = np

        mean_precision_scores = []

        # loop over validation set to evaluate
        for i in validation_range:
            X, y = Xy_preprocess(F3labeled[i], new_thresh)
            if svm.predict(X).sum() == 0:
                print('broken!')
            # focus on precision, as we want highest proportion of tru
            mean_precision = np.mean(cross_val_score(svm, X, y, scorin

            mean_precision_scores.append(mean_precision)

        if np.mean(mean_precision_scores) > best_score:
            best_score = np.mean(mean_precision_scores)
            best_alpha = a
            best_epsilon = e

print(f'Best Precision Score: {best_score}')
print(f'Alpha: {best_alpha}')
print(f'Epsilon: {best_epsilon}')
```

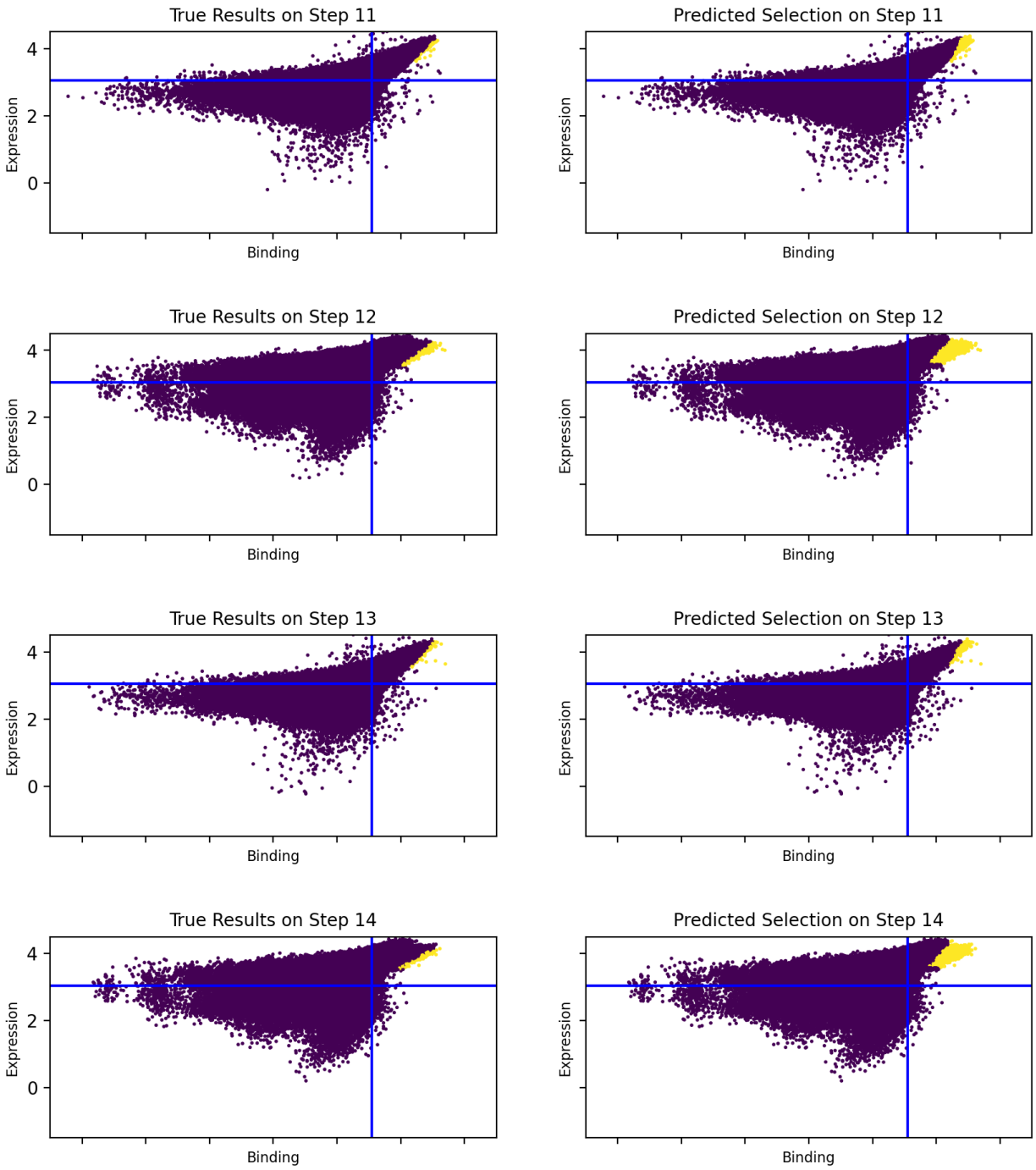
```
Best Precision Score: 0.8120798619119067
Alpha: 1e-06
```

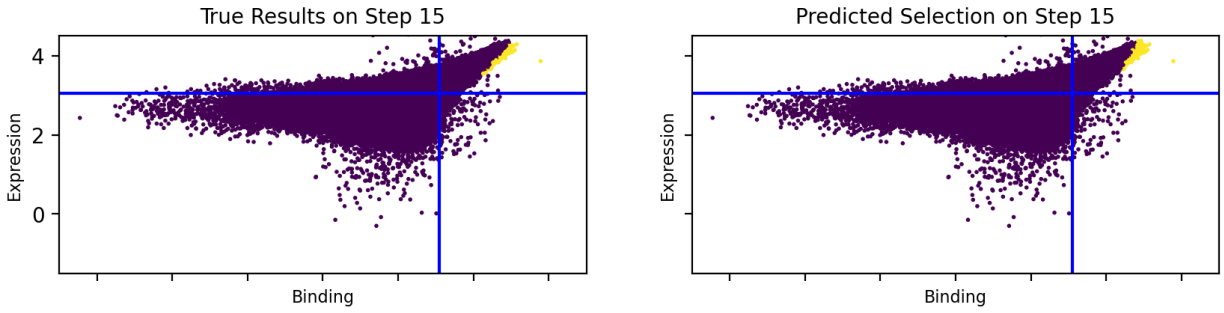
```
Epsilon: 0.1
```

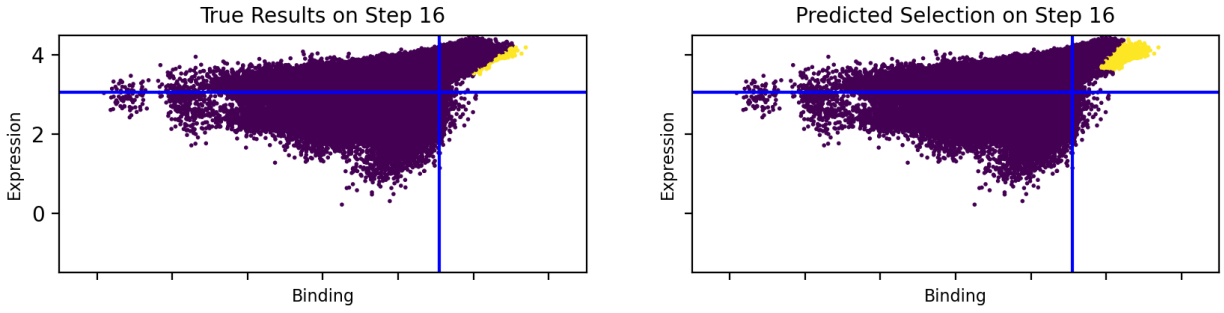
In [10]:
```
# replicate process to fit the model once the hyperparameters are foun
smote = SMOTE(random_state=29)

svm = SGDClassifier(loss='hinge', alpha=best_alpha, random_state=29)
threshold = threshold + best_epsilon

# fit model on training
for frames in [F1labeled, F2labeled]:
    for frame in frames:
        X, y = Xy_preprocess(frame, threshold)

        # oversample response so that prediction can be better
        X_resampled, y_resampled = smote.fit_resample(X, y)

        # update streaming estimator
        svm.partial_fit(X_resampled, y_resampled, classes = np.unique(
```

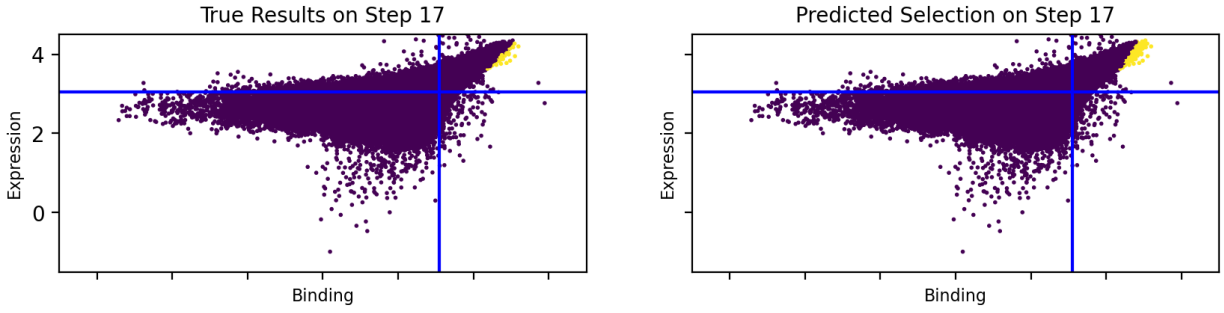
In [11]:
```
fig, axs = plt.subplots(11, 2, figsize = (10, 33), sharey = True, shar

for i in range(11):
    # narrow range to test set
    X_new = F3labeled[11+i][F3labeled[11+i]['AF 488-A'] > threshold][[

    # scale data
    scaler = StandardScaler()
    X_new = scaler.fit_transform(X_new)

    # predict response
    predicted = svm.predict(X_new)

    # identify un-included data
    not_included = F3labeled[11+i][F3labeled[11+i]['AF 488-A'] <= thre
    not_included['predicted'] = 0

    # return results to regular scale
    results = pd.DataFrame({"AF 488-A": X_new[:,0], "AF 647-A": X_new[
    results[['AF 488-A', 'AF 647-A']] = scaler.inverse_transform(resul

    # combine un-included and included data
    results = pd.concat([results, not_included], axis = 0, ignore_inde

    # visualize data
    axs[i,0].scatter(F3labeled[11+i]['AF 647-A'], F3labeled[11+i]['AF
    axs[i,1].scatter(results['AF 647-A'], results['AF 488-A'], s=1, c

    for j in range(2):
        axs[i,j].set_xlim([-2.5, 4.5])
```

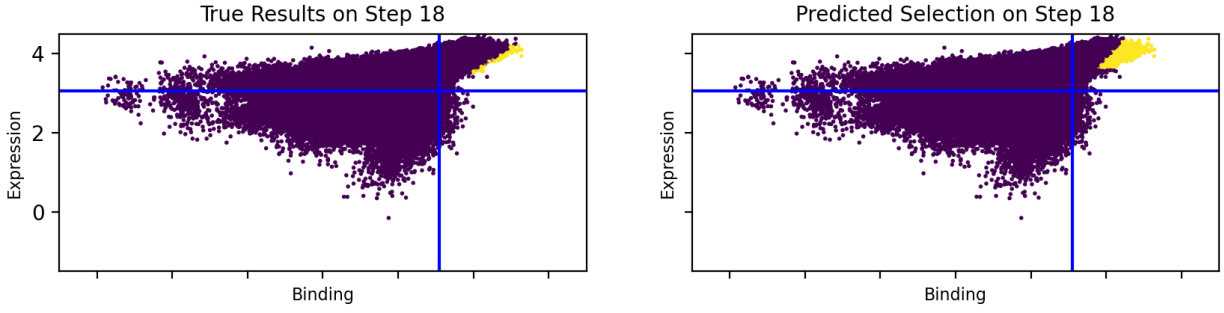
```
        axs[i,j].set_ylim([-1.5, 4.5])
        axs[i,j].set_xlabel("Binding", fontsize = 8)
        axs[i,j].set_ylabel("Expression", fontsize = 8)
        axs[i,j].axhline(y=3.05, c = 'b')
        axs[i,j].axvline(x=2.55, c = 'b')

    axs[i,0].set_title(f"True Results on Step {11+i}", fontsize = 10)
    axs[i,1].set_title(f"Predicted Selection on Step {11+i}", fontsize

plt.subplots_adjust(hspace=0.5)
```

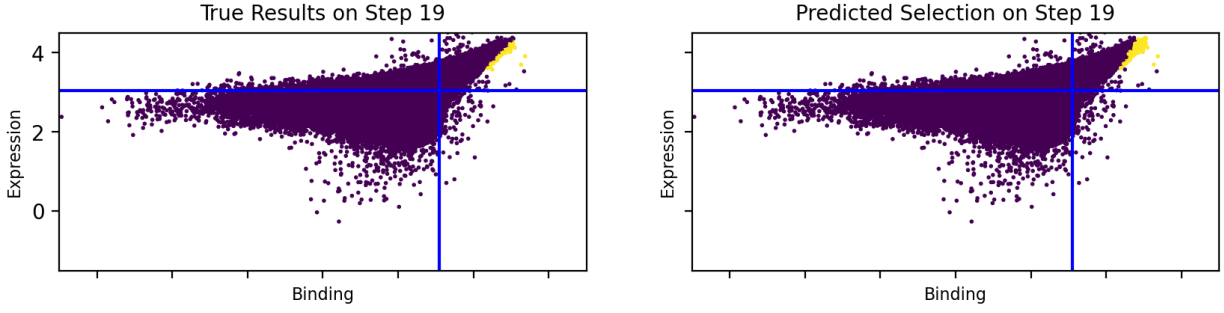



True Results on Step 15
Predicted Selection on Step 15
Expression
Binding
4
2
0

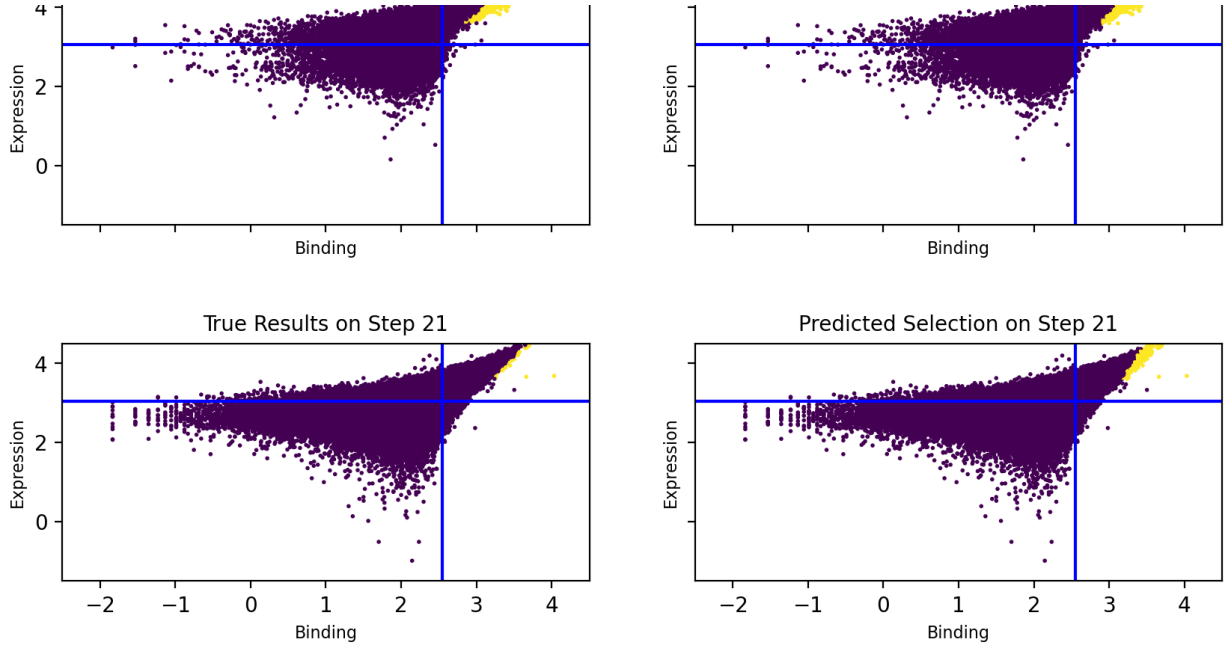

True Results on Step 16
Predicted Selection on Step 16
Expression
Binding
4
2
0


True Results on Step 17
Predicted Selection on Step 17
Expression
Binding
4
2
0


True Results on Step 18
Predicted Selection on Step 18
Expression
Binding
4
2
0


True Results on Step 19
Predicted Selection on Step 19
Expression
Binding
4
2
0


True Results on Step 20
Predicted Selection on Step 20



In [ ]:
```
import joblib

fig.savefig('test_results.jpeg')
joblib.dump(svm, 'DirectedEvolutionSVM.pkl')
```

In [ ]:
```

```