ridge_regression_vs_svm_prob

May 2, 2019

1 Support Vector Machine Vs. Ridge Regression

In this problem, we compare linear SVM and Ridge Regression in the task of classification. As we shall see, formulating the problem as different optimization problems (here SVM and Ridge Regression) makes a difference in performance. There are three places with todos, follow the todos to complete this problem.

In [24]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.linear_model import Ridge
from sklearn.metrics import accuracy_score
```

In [25]:
```
# Helper function for visualization. No todo here.
# Usage: plot_boundry(X, y, fitted_model)
#     X: your features, where each row is a data sample
#     y: your labels, can be 0/1 or -1/1
#     fitted_model: a scipy TRAINED model, such as sklearn.svm.SVC
#
def plot_boundry(X, y, fitted_model):

    plt.figure(figsize=(9.8,5), dpi=100)

    for i, plot_type in enumerate(['Decision Boundary']):
        plt.subplot(1,2,i+1)

        mesh_step_size = 0.5  # step size in the mesh
        x_min, x_max = X[:, 0].min() - .1, X[:, 0].max() + .1
        y_min, y_max = X[:, 1].min() - .1, X[:, 1].max() + .1
        x_max = 110
        y_max = 60
        xx, yy = np.meshgrid(np.arange(x_min, x_max, mesh_step_size), np.arange(y_min
        if i == 0:
            Z = fitted_model.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = np.sign(Z)
```

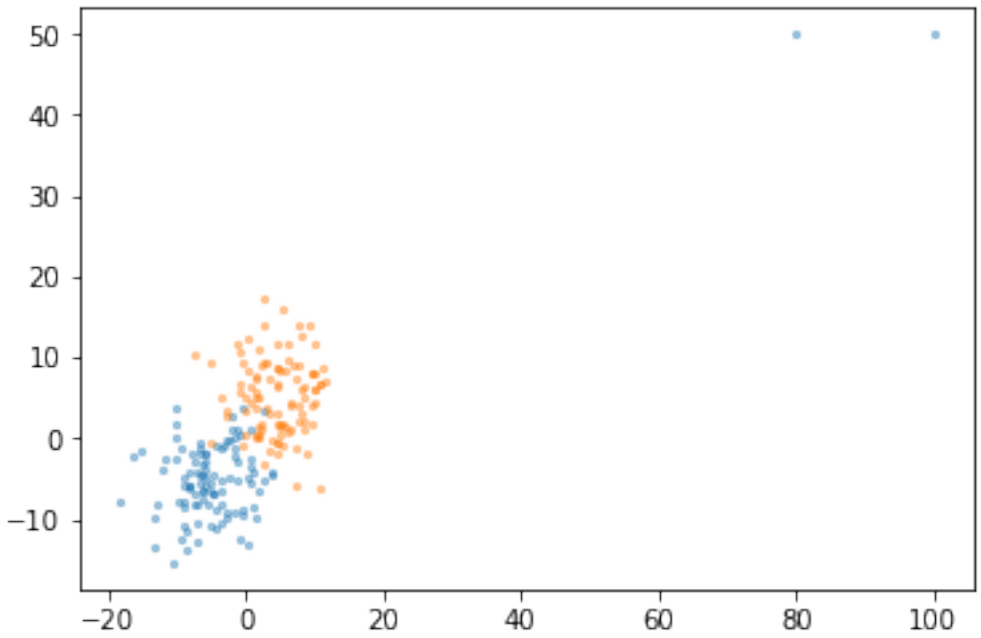
```
            else:
                try:
                    Z = fitted_model.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:,1]
                except:
                    plt.text(0.4, 0.5, 'Probabilities Unavailable', horizontalalignment='
                         verticalalignment='center', transform = plt.gca().transAxes, font
                    plt.axis('off')
                    break
                Z = Z.reshape(xx.shape)
                plt.scatter(X[y==0,0], X[y==0,1], alpha=0.4, label='Edible', s=5)
                plt.scatter(X[y==1,0], X[y==1,1], alpha=0.4, label='Posionous', s=5)
                plt.imshow(Z, interpolation='nearest', cmap='RdYlBu_r', alpha=0.15,
                           extent=(x_min, x_max, y_min, y_max), origin='lower')
                plt.title(plot_type)
                plt.gca().set_aspect('equal');

        plt.tight_layout()
        plt.subplots_adjust(top=0.9, bottom=0.08, wspace=0.02)
```

```
In [26]: # load the data
         train_data = np.load("ridge_vs_svm_data_train.npy")
         X_train = train_data[:, 1:]
         y_train = train_data[:, 0]

         test_data= np.load("ridge_vs_svm_data_train.npy")
         X_test = test_data[:, 1:]
         y_test = test_data[:, 0]
```

Here we visualize the training data to get a sense of the distribution. Note the outliers.

```
In [27]: plt.scatter(X_train[y_train==0,0], X_train[y_train==0,1], alpha=0.4, s=5)
         plt.scatter(X_train[y_train==1,0], X_train[y_train==1,1], alpha=0.4, s=5)
```

```
Out[27]: <matplotlib.collections.PathCollection at 0x22b1c7dd588>
```

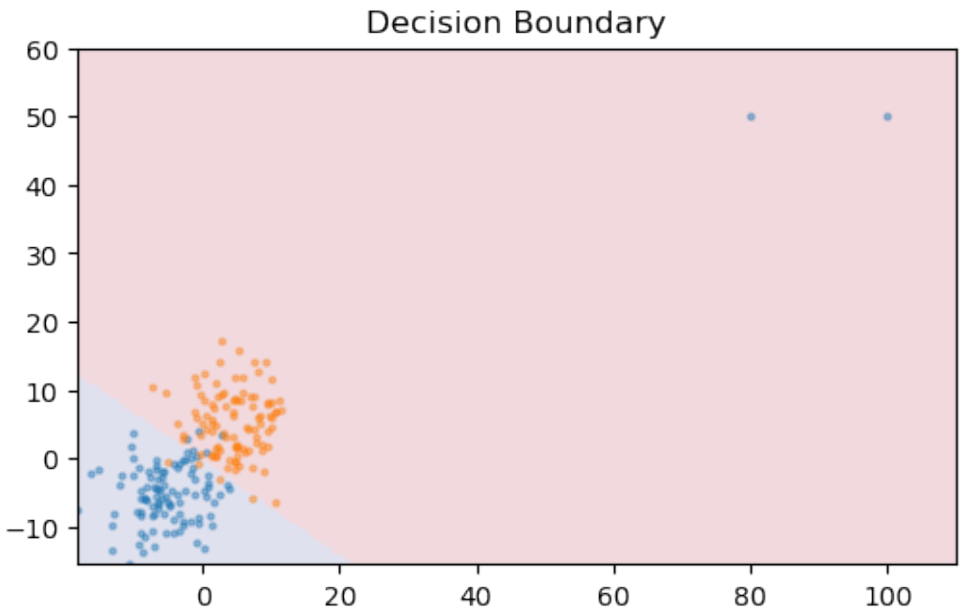


1.1 SVM

Fill in the code below to run a **linear** svm to classify the data.

```
In [28]: fitted_model = SVC(kernel = 'linear').fit(X_train, y_train) # your trained model (as
         y_pred = None # the prediction of your trained model on the testing data

         ########## Your beautiful code starts here ##########
         # todo: Write code to train an SVM, and generate prediction y_pred.
         # Optional: Try using a linear kernel and a polynomial kernel.  How does the value of


         ########## Your beautiful code ends here ##########

         y_pred = fitted_model.predict(X_test)


         accuracy = accuracy_score(y_pred, y_test)
         print("Test Accuracy: {}".format(accuracy))

         plot_boundry(X_test, y_test, fitted_model)
```

```
Test Accuracy: 0.9356435643564357
```

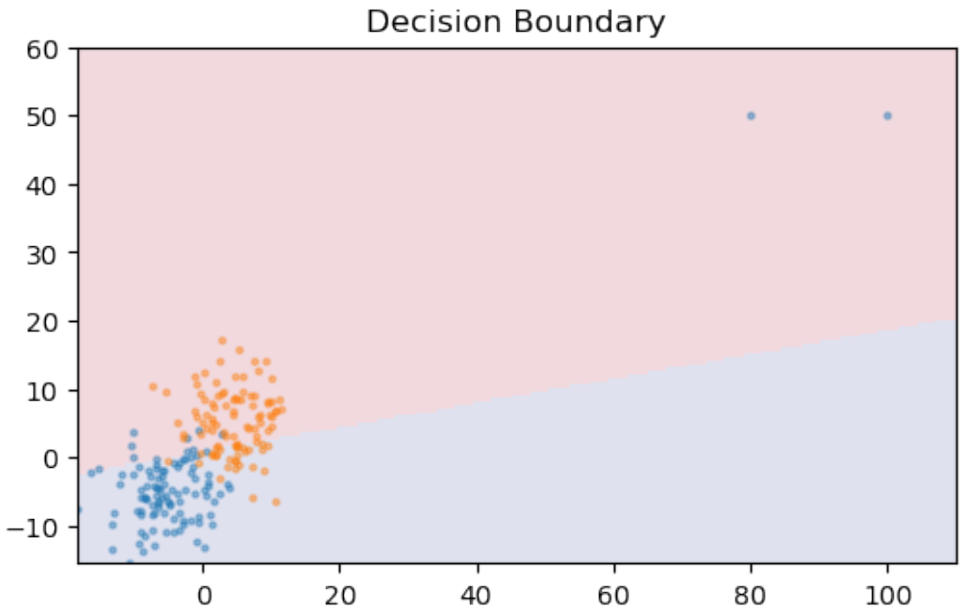


1.2 Ridge Regression

Fill in the code below to run ridge regression to classify the data.

```
In [29]: reg = 0.1
         fitted_model = Ridge(alpha = reg) # your trained model (as trained by scipy)
         y_pred_sign = None # the prediction of your trained model on the testing data

         # convert the labels from 0 and 1 to -1 and 1
         y_train_sign = np.array(y_train)
         y_test_sign = np.array(y_test)
         y_train_sign[y_train_sign == 0] = -1
         y_test_sign[y_test_sign == 0] = -1

         # for the regularization parameter lambda, you can try something around 0.1 :)
         # Optional: try choosing different parameters

         ########## Your beautiful code starts here ##########
         # todo: train a fitted_model and run prediction to generate y_pred_sign
         fitted_model.fit(X_train, y_train_sign)
         y_pred_sign = fitted_model.predict(X_test)
         y_pred_sign = np.where(y_pred_sign < 0, -1, 1)
         ########## Your beautiful code ends here ##########
         accuracy = accuracy_score(y_pred_sign, y_test_sign)
         print("Test Accuracy: {}".format(accuracy))

         plot_boundry(X_test, y_test, fitted_model)
```

```
Test Accuracy: 0.8168316831683168
```



2 Why Do We See SVM Outperforming Ridge Regression?

In the above, we saw that SVM outperforms ridge regression because SVM is more robust to outliers. The data was actually synthetically generated from two Gaussians --- but remember the two outliers? Can you see how they are impacting the classifer?

2.1 The support vector classifier's separating hyperplane is only dependent on the support vectors for the given dataset, while ridge regression's separating hyperplane is heavily influenced by the location of the outliers and will seek to fit to the outliers much more than the SVM will.

3 How the Data was produced

In [30]:
```
# Optional: Try changing the positions of the outliers to see how they impact the per

n = 100
cov = np.eye(2) * 20

pos = np.hstack([
    np.ones(n).reshape([-1, 1]),
    np.random.multivariate_normal([5, 5], cov, size=n),
])
neg = np.hstack([
```

```
    np.zeros(n).reshape([-1, 1]),
    np.random.multivariate_normal([-5, -5], cov, size=n),
])

syn = np.vstack([pos, neg])

outliers = np.array([
    [0, 80, 50,],
    [0, 100, 50,],
])
syn = np.vstack([pos, neg, outliers])
np.random.shuffle(syn)
np.save("ridge_vs_svm_data_train.npy", syn)


pos_test = np.hstack([
    np.ones(n).reshape([-1, 1]),
    np.random.multivariate_normal([5, 5], cov, size=n),
])
neg_test = np.hstack([
    np.zeros(n).reshape([-1, 1]),
    np.random.multivariate_normal([-5, -5], cov, size=n),
])

syn = np.vstack([pos_test, neg_test])

np.random.shuffle(syn)
np.save("ridge_vs_svm_data_test.npy", syn)
```

4 Credit

Spring 2019: Mong H. Ng, Prof. Ranade Plotting function from https://github.com/devssh/svm/blob/master/SVM%20Python/Classifier%20Visualization.ipynb

pca_descent

May 2, 2019

0.1 PCA Descent

In this problem, we will explore how we can obtain the top eigenvector using a gradient-descent-like method.

In [84]:
```
# It's dangerous to go alone! Take this:
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
%matplotlib inline
```

In [85]:
```
# generating the covariance matrix for the data

# singular vectors, scaled by singular values
v1 = np.array([1, 3]).reshape(2, 1)
v2 = np.array([-3 / 4, 1 / 4]).reshape(2, 1)

# sum of dyads
cov = v1 @ v1.T + v2 @ v2.T
print("Covariance matrix:\n", cov, '\n')
# TODO:  What do you expect the first principal component
# of the covariance matrix to be? (see v1, v2)

print('Here are the principal components of the covariance matrix:\n')
U, s, V = np.linalg.svd(cov)
print(U)
```

```
Covariance matrix:
 [[1.5625 2.8125]
 [2.8125 9.0625]]

Here are the principal components of the covariance matrix:

[[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]
```

In [86]:
```
# generating the data from the covariance matrix
```

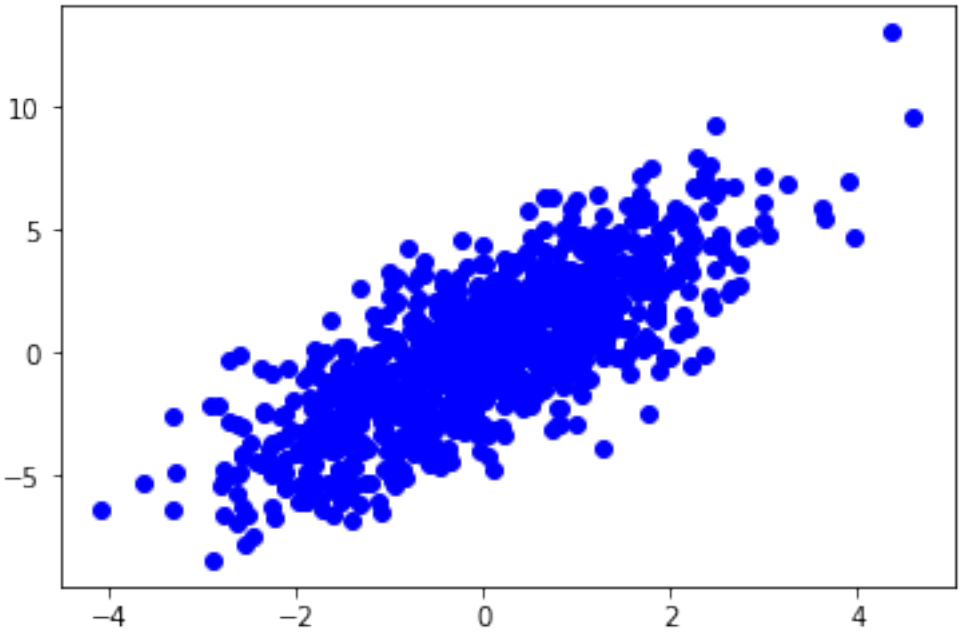
```
        # normally distributed with mean 0 and covariance `cov`
        N = 1000
        # chugga chugga choo choo
        data = np.random.multivariate_normal(mean=np.zeros(2), cov=cov, size=N)
```

In [87]:
```
data.shape
```

Out[87]:
```
(1000, 2)
```

In [88]:
```
# plot the data!
plt.scatter(data[:, 0], data[:, 1], color="blue")
```

Out[88]:
```
<matplotlib.collections.PathCollection at 0x2883cb8b630>
```



Now let's try to implement the gradient rule. Recall that we are updating according to:

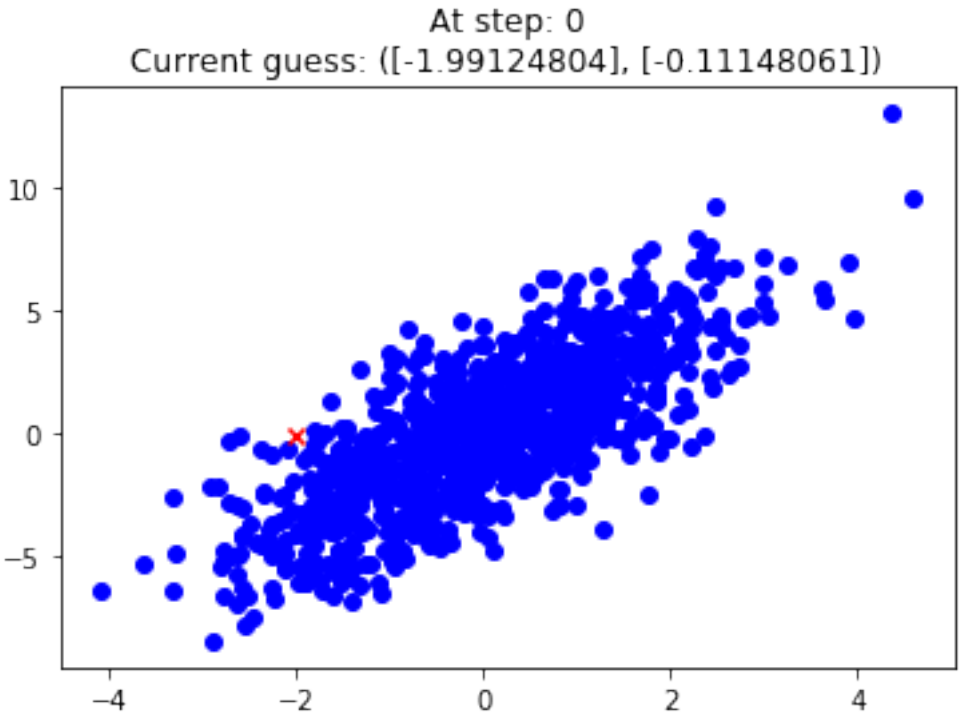
$$Y_k = Y_{k-1} + \gamma \left(X_k X_k^\top - \frac{Y_{k-1}^\top X_k X_k^\top Y_{k-1}}{||Y_{k-1}||^2} \right) Y_{k-1}$$

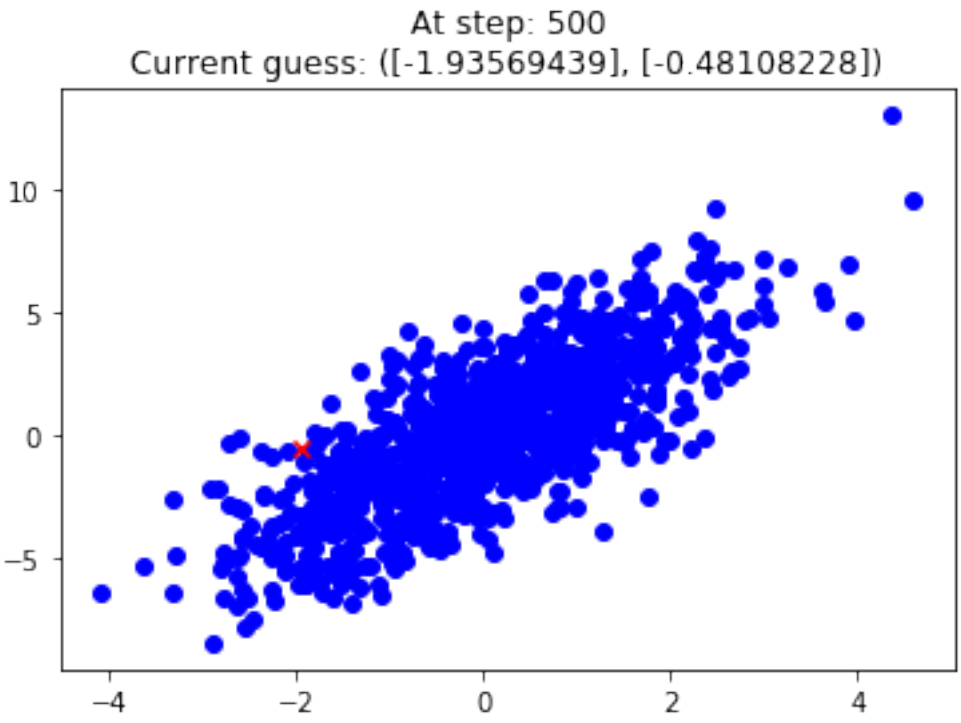
where Y is our "guessed" vector corresponding to the maximal singular value and X_k is our k-th datapoint that we are using (notice that this will be the same as the timestep k).

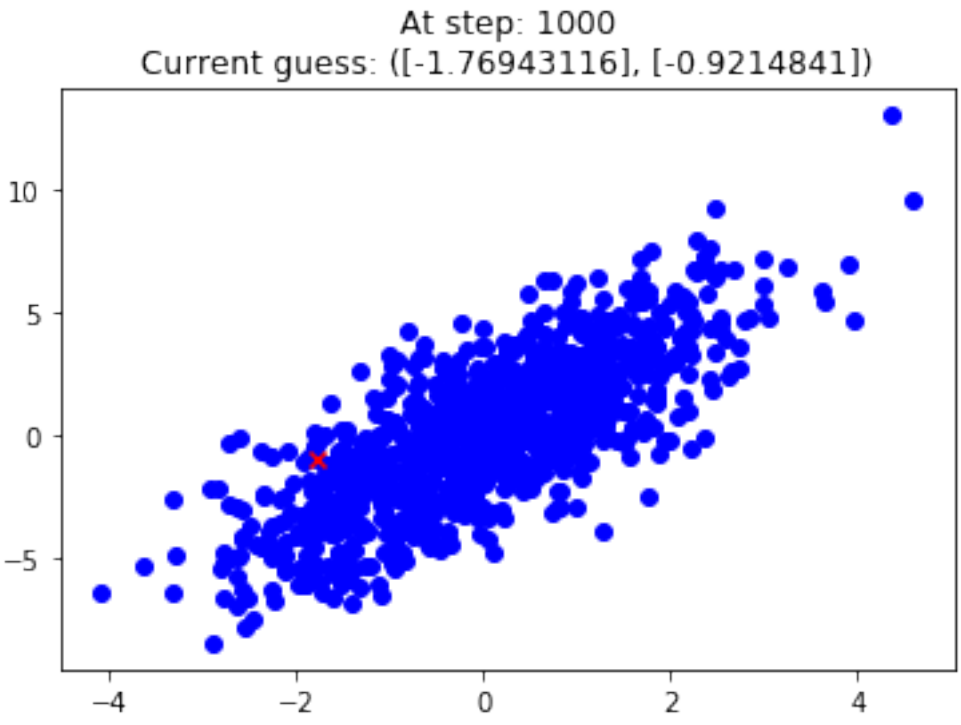
In [89]:
```
# defining the stochastic descent (well, let's hope it's descent) function
# see the gradient update rule outlined in part c, it's about to get hairy

# defining the expressions outlined in
```

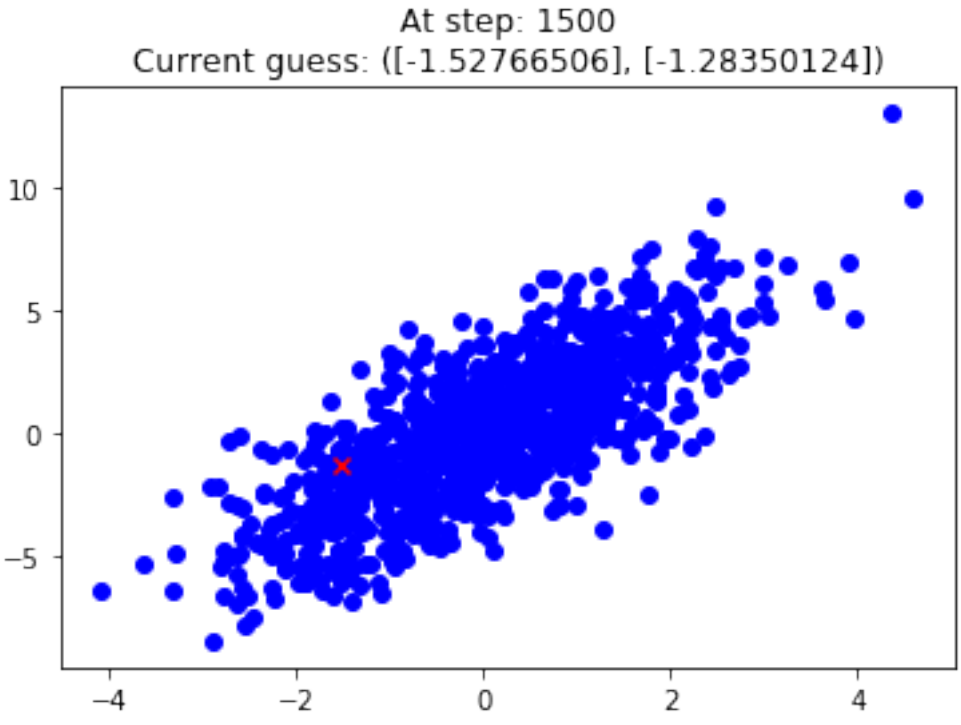
```
    """T. P. Krasulina, A method of stochastic approximation for
    the determination of the least eigenvalue of a symmetric
    matrix, Zh. Vychisl. Mat. Mat. Fiz., 1969, Volume 9,
    Number 6, 13831387 """
    def update(x, y, LR):
        """Computes the Krasulina update.

        Args:
          x: numpy array, a given data point from `data`
          y: numpy array, the current guess for the "maximal" singular vector
          LR: float, the learning rate

        Returns:
          y_next: numpy array, the new guess for the "maximal" singular vector
        """
        x = x.reshape(2, 1)
        y = y.reshape(2, 1)

        # Fill in the update rule to return the next step in the stochastic gradient desc
        gradient = ((x @ x.T) - ((y.T @ x @ x.T @ y) * np.eye(2) / (np.linalg.norm(y) ** 2
        return y + (LR * gradient)
```

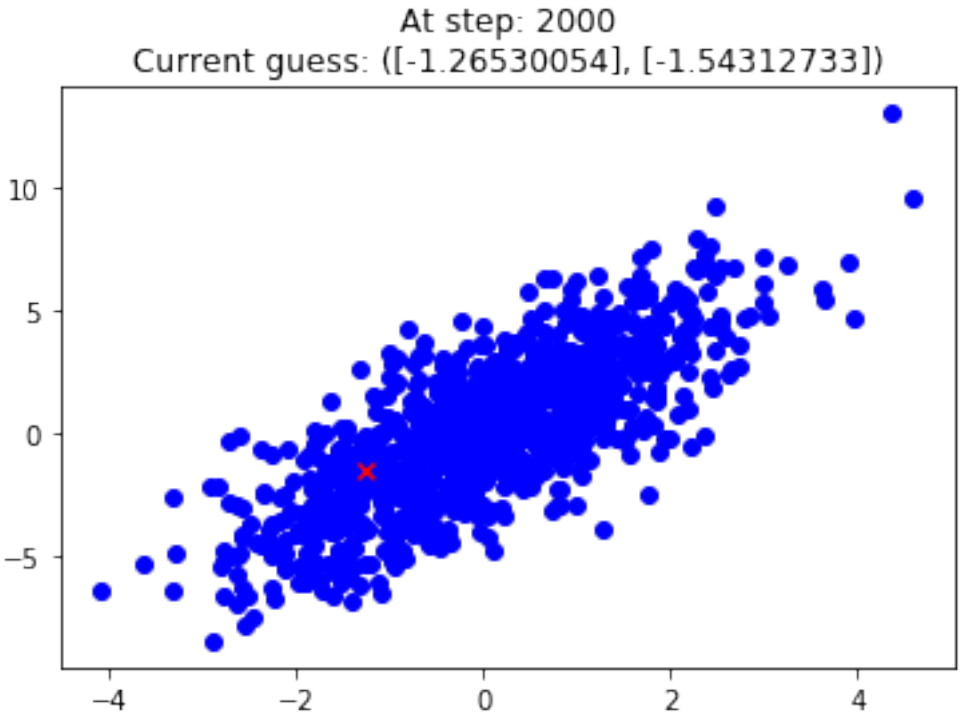
In [95]:
```
    # now, to pray that our ball rolls down the hill

    # initialize y_0
    y = np.random.multivariate_normal(np.zeros(2), 3 * np.eye(2))
    LR = 0.0001  # how did I pick this? I sampled a random ML paper from arXiv.
    num_steps = 10000  # same tbh

    for step in range(num_steps):
        # select the step % Nth point (cycles through the data)
        x = data[step % N]
        y = update(x, y, LR)

        # display every 1/20th of the process
        if step % (num_steps // 20) == 0:
            # plot original data
            plt.scatter(data[:, 0], data[:, 1], color="blue")

            # plot current guess
            plt.scatter([y[0]], [y[1]], color="red", marker="x")

            plt.title("At step: {}\nCurrent guess: ({}, {})".format(
                    step, y[0], y[1]))
            plt.show()
```

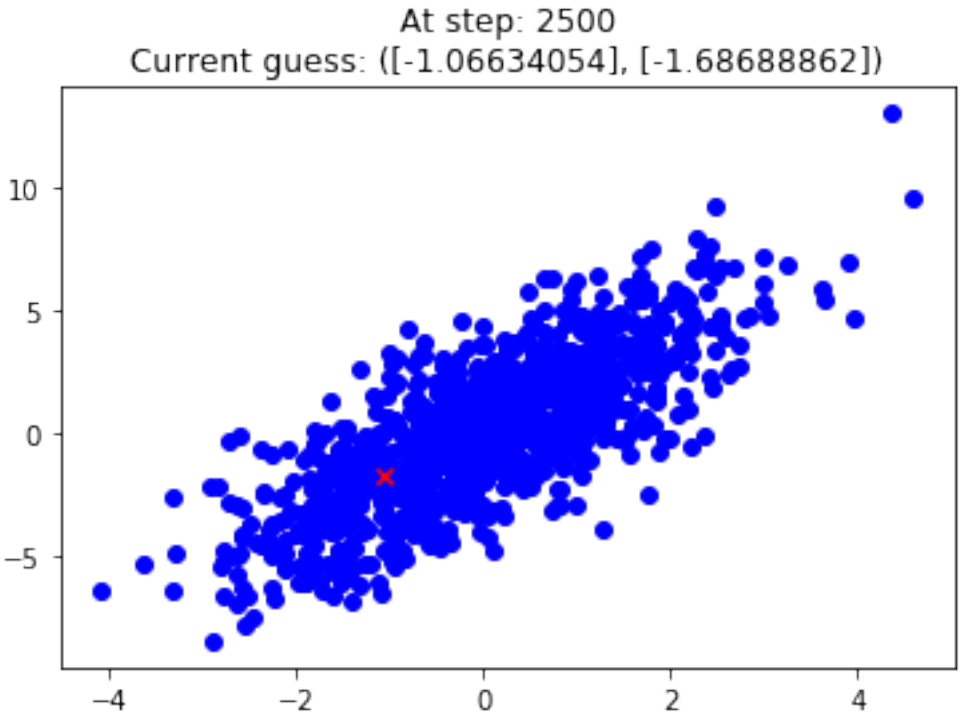

At step: 0
Current guess: ([-1.99124804], [-0.11148061])
10
5
0
−5
−4
−2
0
2
4

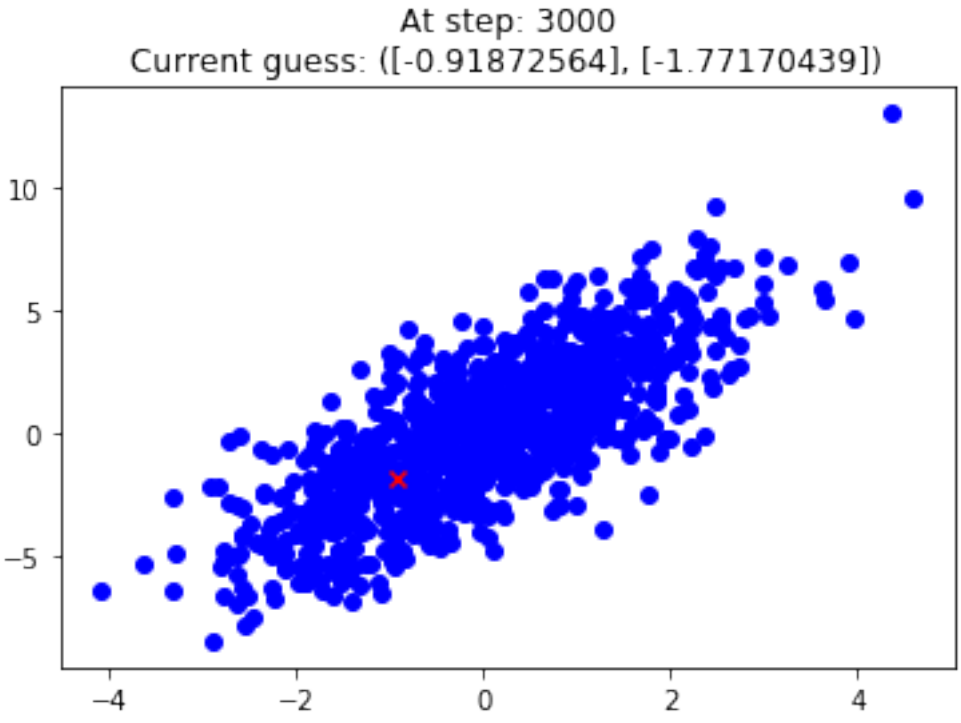

At step: 500
Current guess: ([-1.93569439], [-0.48108228])
10
5
0
−5
−4
−2
0
2
4

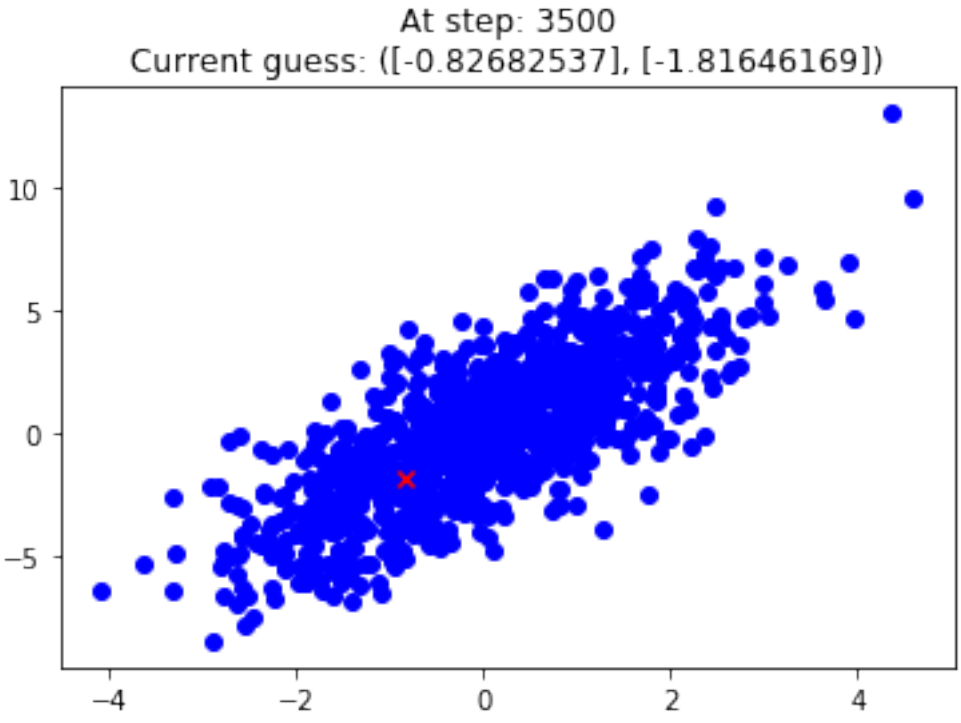

At step: 1000
Current guess: ([-1.76943116], [-0.9214841])
10
5
0
−5
−4
−2
0
2
4

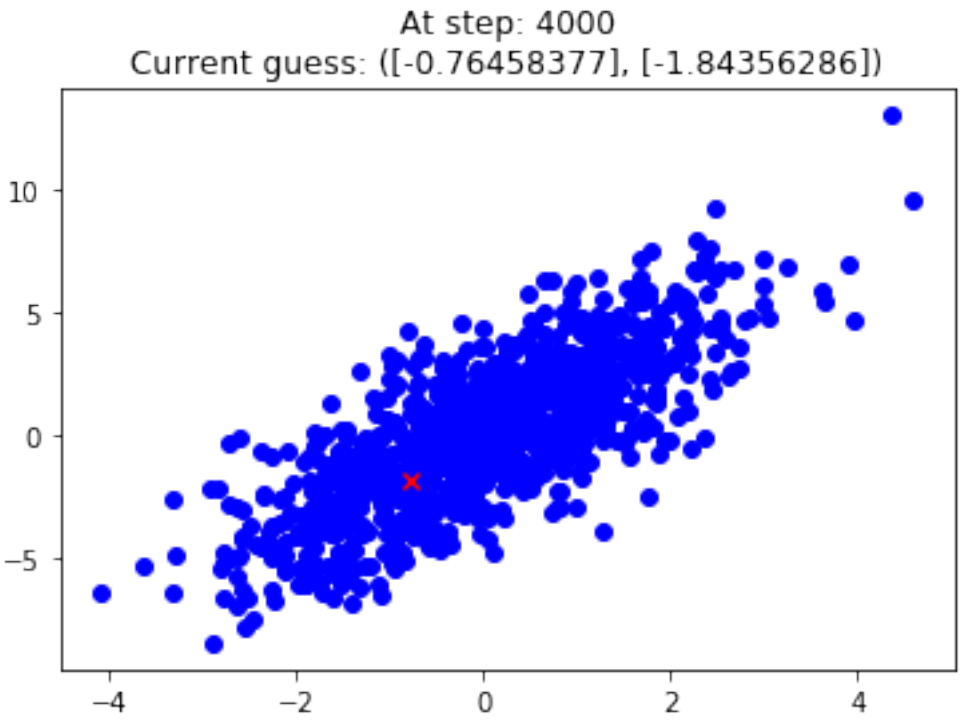

At step: 1500
Current guess: ([-1.52766506], [-1.28350124])
10
5
0
−5
−4
−2
0
2
4

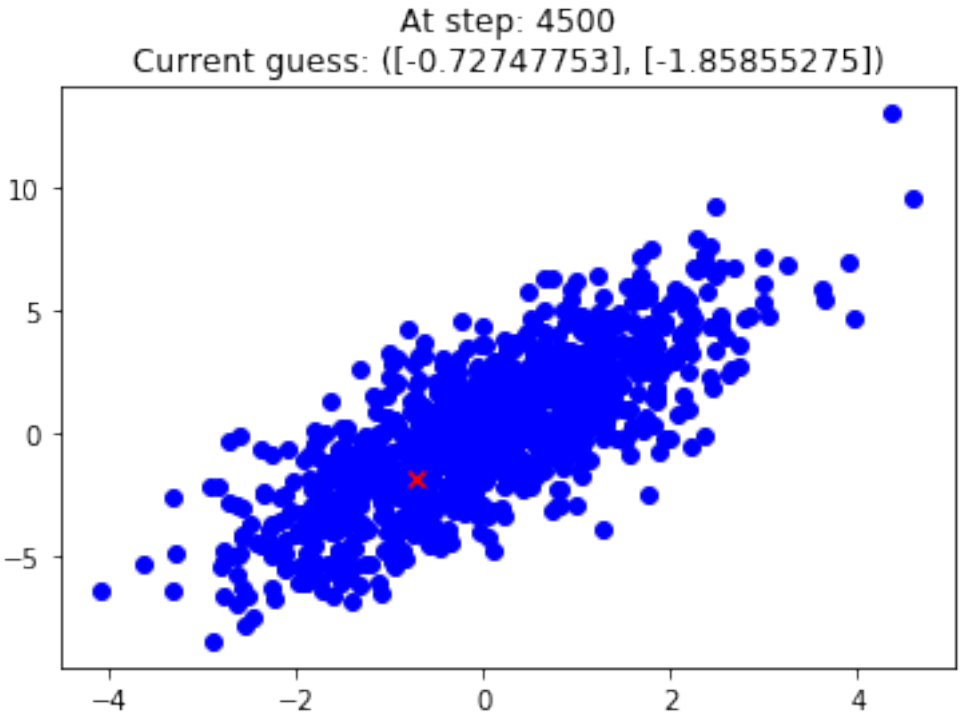

At step: 2000
Current guess: ([-1.26530054], [-1.54312733])
10
5
0
−5
−4
−2
0
2
4

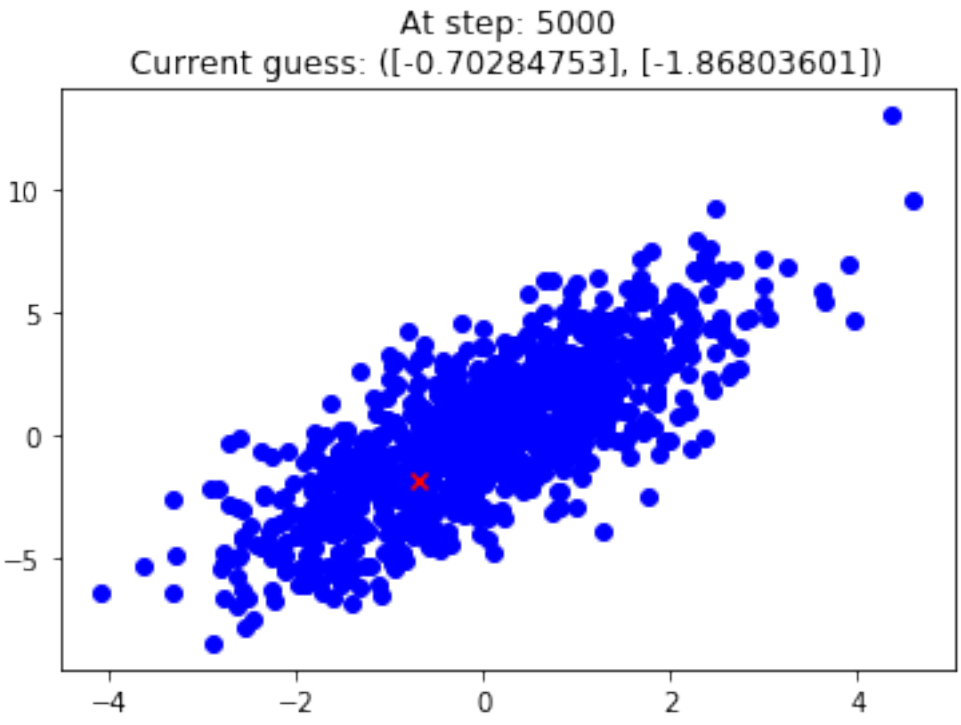

At step: 2500
Current guess: ([-1.06634054], [-1.68688862])
10
5
0
−5
−4
−2
0
2
4

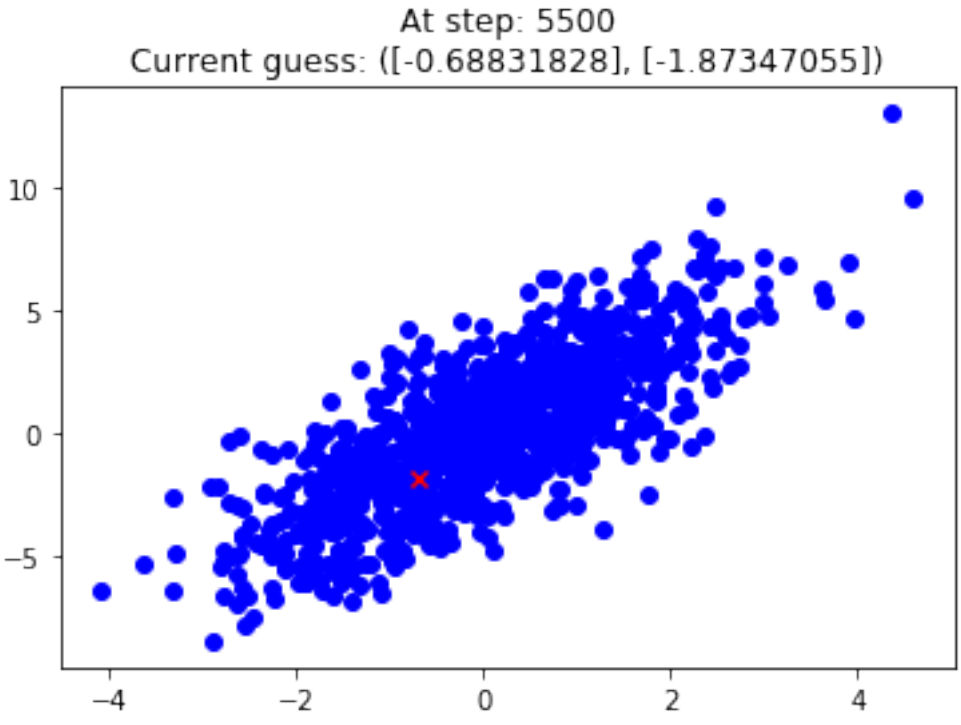

At step: 3000
Current guess: ([-0.91872564], [-1.77170439])
10
5
0
−5
−4
−2
0
2
4

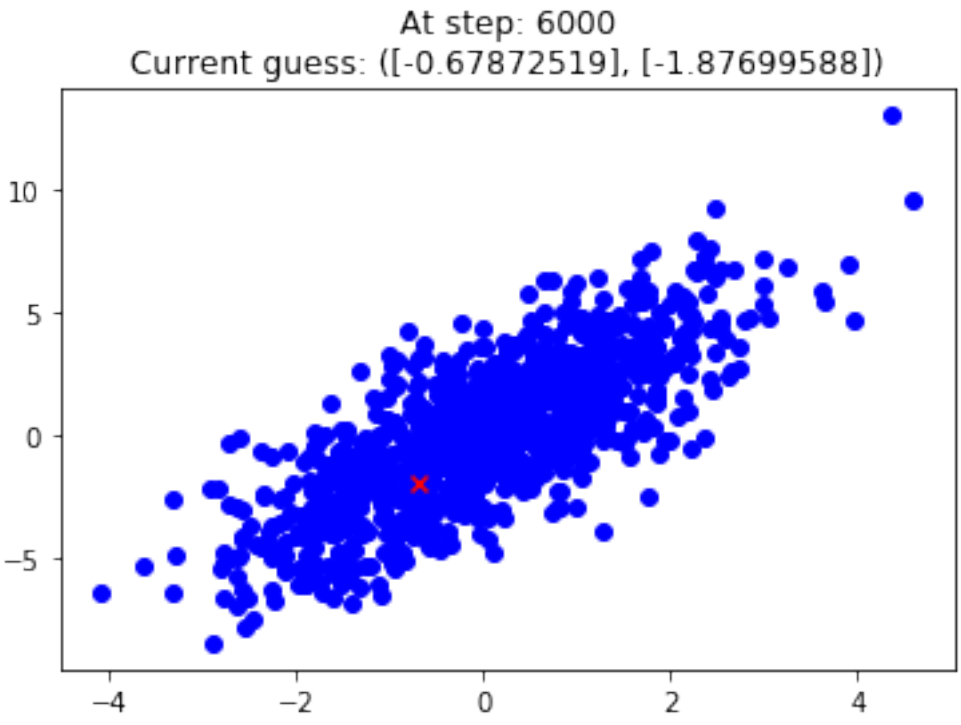

At step: 3500
Current guess: ([-0.82682537], [-1.81646169])
10
5
0
−5
−4
−2
0
2
4

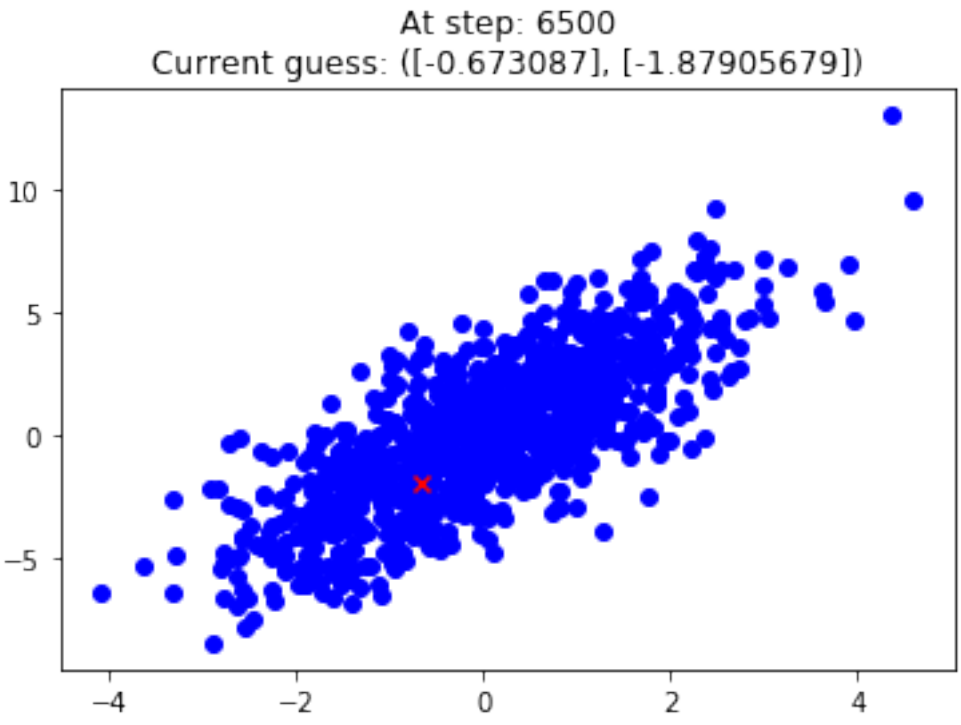

At step: 4000
Current guess: ([-0.76458377], [-1.84356286])
10
5
0
−5
−4
−2
0
2
4

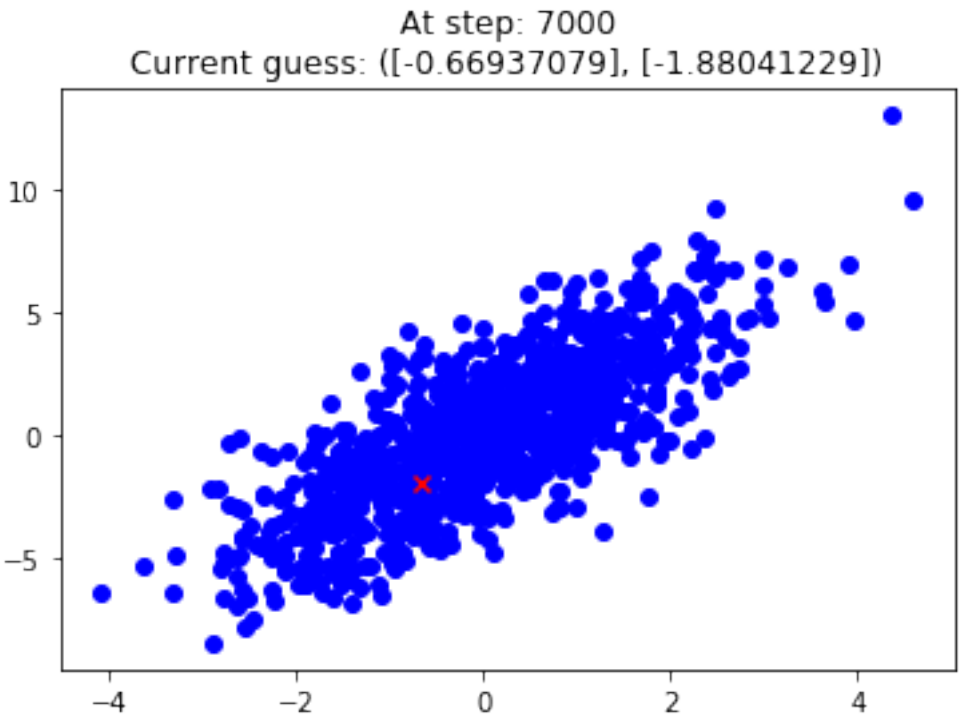

At step: 4500
Current guess: ([-0.72747753], [-1.85855275])
10
5
0
−5
−4
−2
0
2
4

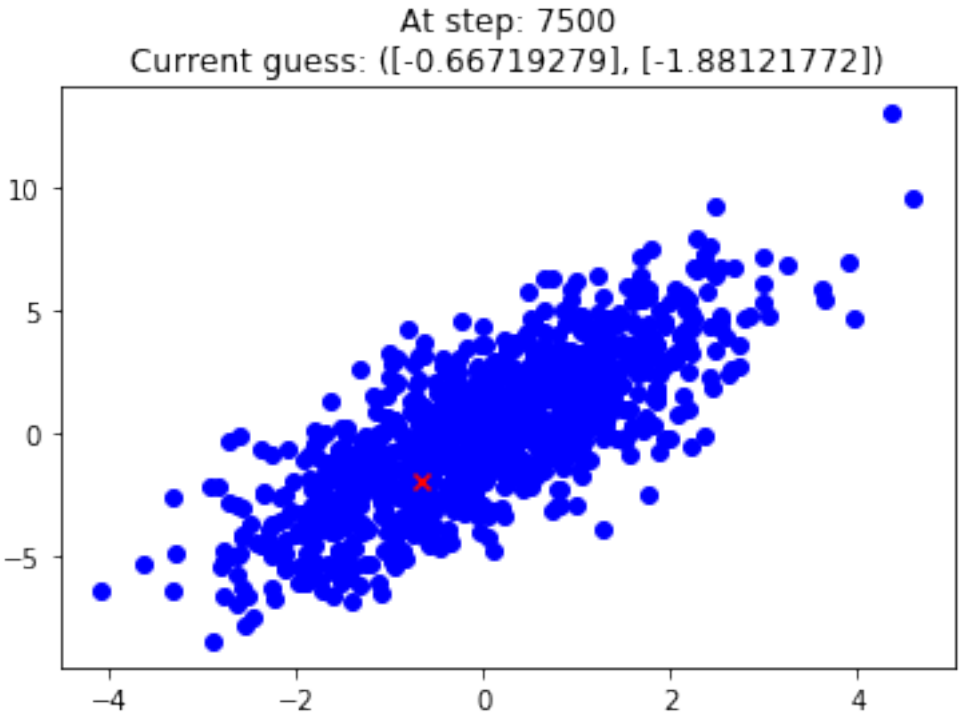

At step: 5000
Current guess: ([-0.70284753], [-1.86803601])
10
5
0
−5
−4
−2
0
2
4

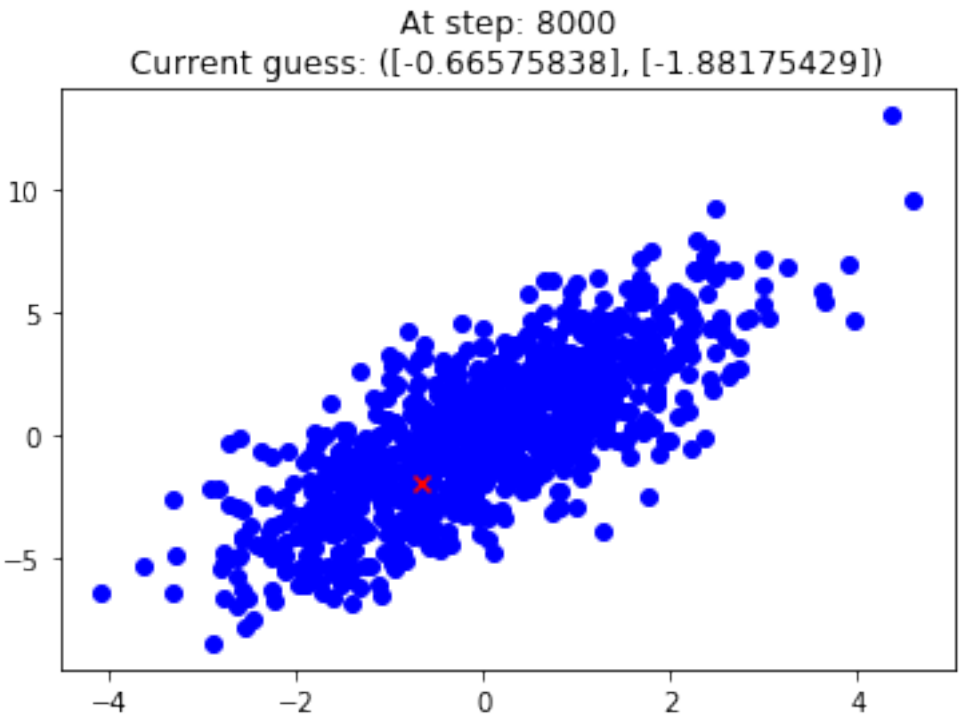

At step: 5500
Current guess: ([-0.68831828], [-1.87347055])
10
5
0
−5
−4
−2
0
2
4

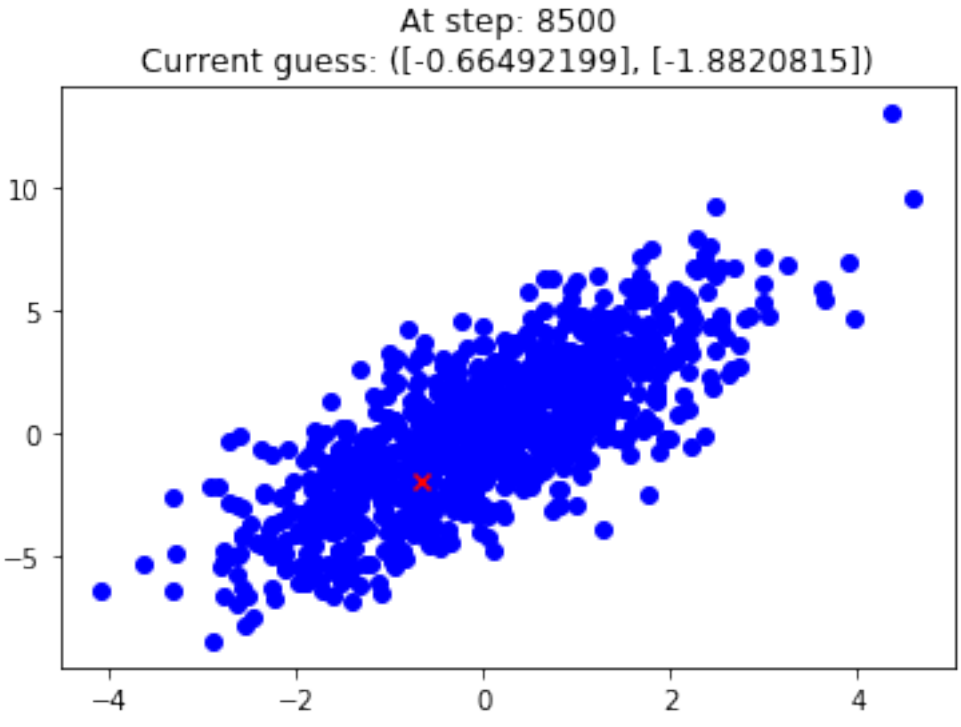

At step: 6000
Current guess: ([-0.67872519], [-1.87699588])
10
5
0
−5
−4
−2
0
2
4

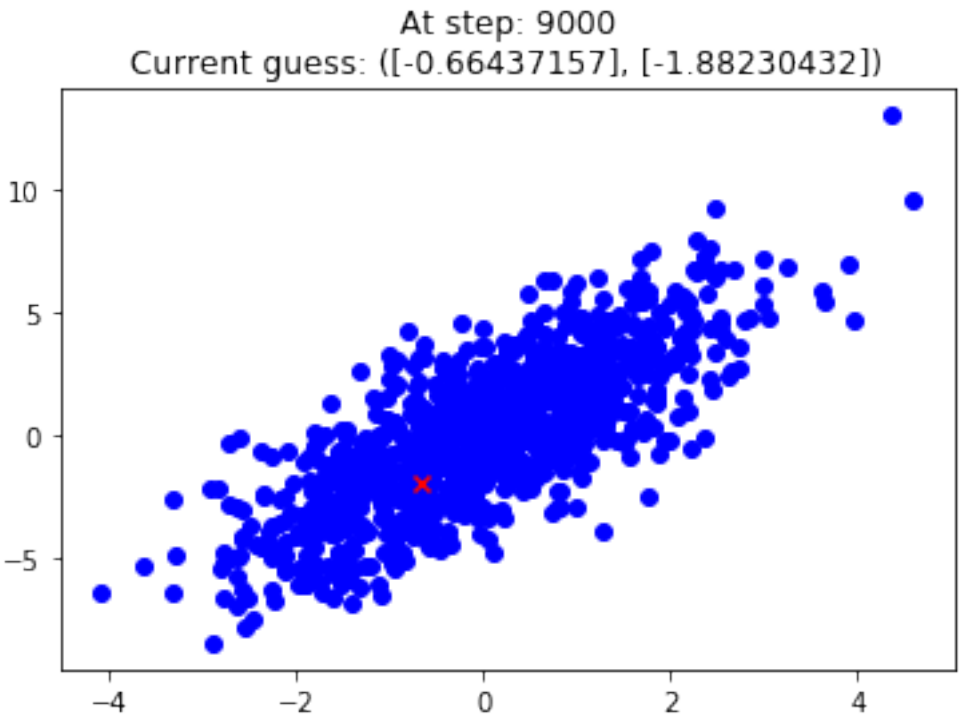

At step: 6500
Current guess: ([-0.673087], [-1.87905679])
10
5
0
−5
−4
−2
0
2
4

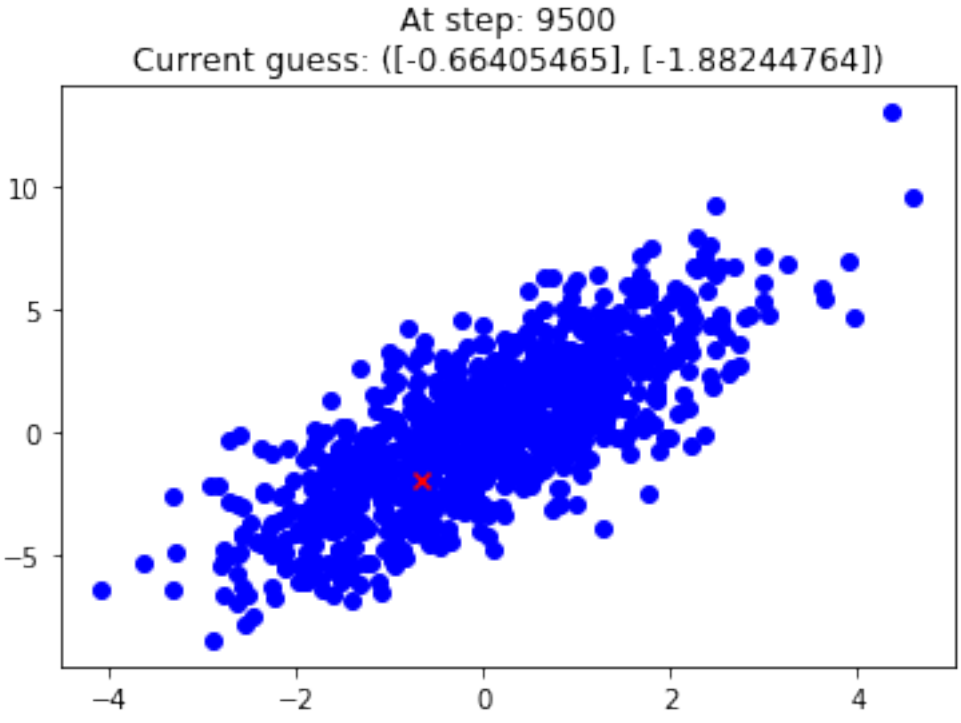

At step: 7000
Current guess: ([-0.66937079], [-1.88041229])
10
5
0
−5
−4
−2
0
2
4


At step: 7500
Current guess: ([-0.66719279], [-1.88121772])
10
5
0
−5
−4
−2
0
2
4


At step: 8000
Current guess: ([-0.66575838], [-1.88175429])
10
5
0
−5
−4
−2
0
2
4


At step: 8500
Current guess: ([-0.66492199], [-1.8820815])
10
5
0
−5
−4
−2
0
2
4





Check that the final vector is along the same axis as the vector you were expecting!

Normalize the output vector that we get so that we can compare it to the principal component of the covariance matrix

```
In [96]: normalized_y = y/np.linalg.norm(y)
         print(normalized_y)
```

```
[[-0.33262831]
 [-0.94305801]]
```

```
In [97]: print(U[0], " I am comparing this")
```

```
[-0.31622777 -0.9486833 ]  I am comparing this
```

hydroelectric_power

May 2, 2019

1 Hydroelectric Power Generation

```
In [83]: import numpy as np
         import cvxpy as cp
```

1.1 Part 1

```
In [84]: K = 3
         L = 2
         J = 3
         w_l = np.array([20, 15])
         lam = np.array([10, 30, 8])
         rho = np.array([30./47, 43./58, 50./72])
         V_0 = np.array([120, 130])
         V_min = np.zeros((L, K))
         V_max = np.array([[95000, 95000, 95000], [11000, 11000, 11000]])
         A = np.array([[15, 15, 15], [12, 12, 12]])
         T_max = np.array([[55, 55, 55], [65, 65, 65], [80, 80, 80]])
         T_min = np.zeros((J, K))
```

```
In [104]: def solve(K, L, J, w_l, lam, rho, V_0, V_min, V_max, A, T_min, T_max):
              """Solve the optimization problem described in part 1 using a solver of your cho

              Args:
                  K (int): number of timestesp
                  L (int): number of reservoirs
                  J (int): number of turbines
                  etc.

              Returns:
                  the optimal water volumes, T_opt
                  the optimal objective value, obj_val
              """
              # TODO: your implementation here
              #print(w_l.shape, lam.shape, rho.shape, V_0.shape, V_min.shape, V_max.shape, A.s
              #print("w_l, lam, rho, V_0, V_min, V_max, A, T_min, T_max")
              V = cp.Variable((L, K))
```

```
    T = cp.Variable((J, K))
    cost = w_l.T @ V_0  - rho.T @ T @ lam - w_l.T @ V[:,K - 1]
    constraints = [T_min <= T,
                   T <= T_max,
                   V_min <= V,
                   V <= V_max,
                   V[0,0] == V_0[0] + A[0,0] - (T[0,0] + T[1,0]),
                   V[1,0] == V_0[1] + A[1,0] + (T[0,0] + T[1,0]) - T[2,0],
                   V[0,1] == V[0,0] + A[0,1] - (T[0,1] + T[1,1]),
                   V[1,1] == V[1,0] + A[1,1] + (T[0,1] + T[1,1]) - T[2,1],
                   V[0,2] == V[0,1] + A[0,2] - (T[0,2] + T[1,2]),
                   V[1,2] == V[1,1] + A[1,2] + (T[0,2] + T[1,2]) - T[2,2],
                   ]
    problem = cp.Problem(cp.Minimize(cost), constraints)
    problem.solve()
    T_opt = T.value
    obj_val = problem.value
    return T_opt, obj_val
```

In [105]:
```
# Solve the optimization problem
T_opt, obj_val = solve(K, L, J, w_l, lam, rho, V_0, V_min, V_max, A, T_min, T_max)

# Round values for readability (optional)
#obj_val = np.around(obj_val, decimals=7)
#T_opt = np.around(T_opt, decimals=0)

# Print results
print("Optimal value:\n{}\n".format(obj_val))
print("Optimal water volumes, T_opt:\n{}".format(T_opt))
```

```
Optimal value:
-3891.92712154561

Optimal water volumes, T_opt:
[[1.68613701e-21 5.50000000e+01 1.45716774e-22]
 [3.00000000e+01 6.50000000e+01 1.50000000e+01]
 [1.17128387e-20 8.00000000e+01 4.71843367e-21]]
```

1.2 Part 2 (Don't forget to re-initialize variables!)

In [106]:
```
K = 3
L = 2
J = 3
w_l = np.array([20, 15])
lam = np.array([12, 30, 4])
rho = np.array([30./47, 43./58, 50./72])
V_0 = np.array([120, 130])
```

```
V_min = np.zeros((L, K))
V_max = np.array([[95000, 95000, 95000], [11000, 11000, 11000]])
A = np.array([[15, 15, 15], [12, 12, 12]])
T_min = np.zeros((J, K))
T_max = np.array([[55, 55, 55], [65, 65, 65], [80, 80, 80]])
```

In [109]:
```
def solve_regularized(K, L, J, T_hat, w_l, lam, rho, V_0, V_min, V_max, A, T_min, T_
    """Solve the optimization problem described in part 2 using a solver of your cho

    Args:
        T_hat (numpy.ndarray): optimal solution T_opt from part 1
        gamma (float): regularization multiplier
        (remaining variables have same description as in part 1)

    Returns:
        the optimal water volumes, T_opt_reg
        the optimal objective value, obj_val_reg
    """
    # TODO: your implementation here
    V = cp.Variable((L, K))
    T = cp.Variable((J, K))
    diff = cp.norm(T - T_hat, 1)
    cost = w_l.T @ V_0  - rho.T @ T @ lam - w_l.T @ V[:,K - 1] + gamma * diff
    constraints = [T_min <= T,
                   T <= T_max,
                   V_min <= V,
                   V <= V_max,
                   V[0,0] == V_0[0] + A[0,0] - (T[0,0] + T[1,0]),
                   V[1,0] == V_0[1] + A[1,0] + (T[0,0] + T[1,0]) - T[2,0],
                   V[0,1] == V[0,0] + A[0,1] - (T[0,1] + T[1,1]),
                   V[1,1] == V[1,0] + A[1,1] + (T[0,1] + T[1,1]) - T[2,1],
                   V[0,2] == V[0,1] + A[0,2] - (T[0,2] + T[1,2]),
                   V[1,2] == V[1,1] + A[1,2] + (T[0,2] + T[1,2]) - T[2,2],
                  ]
    problem = cp.Problem(cp.Minimize(cost), constraints)
    problem.solve()
    T_opt = T.value
    obj_val = problem.value
    return T_opt, obj_val
```

In [110]:
```
# Solve the optimization problem
T_opt_reg, obj_val_reg = solve_regularized(K, L, J, T_opt, w_l, lam, rho, V_0, V_min

# Round values for readability (optional)
obj_val_reg = np.around(obj_val_reg, decimals=7)
T_opt_reg = np.around(T_opt_reg, decimals=0)

# Print results
```

```
        print("Optimal value:\n{}\n".format(obj_val_reg))
        print("Optimal water volumes, T_opt_reg:\n{}".format(T_opt_reg))
```

```
Optimal value:
-3891.9271215

Optimal water volumes, T_opt_reg:
[[ 0. 55.  0.]
 [30. 65. 15.]
 [ 0. 80.  0.]]
```

UNIVERSITY OF CALIFORNIA, BERKELEY

Homework 12

Oscar Ortega

May 3, 2019

1 LASSO VS. RIDGE

- a: Let $X = \begin{bmatrix} x_1 & \dots & x_d \end{bmatrix}$

$$= \min_w \|Xw - y\|_2^2 + \lambda \|w\|_1 \tag{1.1}$$

$$= \min_w w^T X^T X w - 2y^T X w + y^T y + \sum_{i=1}^{d} |w_i| \tag{1.2}$$

$$= \min_w n w^T w - 2y^T X w + y^T y + \lambda \sum_{i=1}^{d} |w_i| \tag{1.3}$$

$$= \min_{w_1,\dots,w_d \in \mathbf{R}} \sum_{i=1}^{d} (n w_i^2 - 2y_i^T X_i w_i + \lambda |w_i|) + y^T y \tag{1.4}$$

- b: Let $w_i^* > 0, L = \|Xw - y\|_2^2 + \lambda \|w\|_1$

$$\frac{\partial L}{\partial_{w_i}} = 2n w_i - 2y^T X_i + \lambda = 0 \tag{1.5}$$

$$w_i^* = \frac{y^T X_i}{n} - \frac{\lambda}{2n} > 0 \tag{1.6}$$

$$y^T X_i > \frac{\lambda}{2} \tag{1.7}$$

- c: Let w_i^*

$$\frac{\partial L}{\partial w_i} = 2nw_i - 2y^T X_i - \lambda \tag{1.8}$$

$$w_i^* = \frac{\lambda}{2n} - \frac{X_i^T y}{n} < 0 \tag{1.9}$$

$$\frac{\lambda}{2} < X_i^T y \tag{1.10}$$

- d: We can conclude that if $|y^T x_i| \leq \frac{\lambda}{2}$ than the value of w_i^* will be non-zero. The value of lambda impacts the threshold that determines whether the the value of w_i^* will be cutoff.

- e:

$$\min_{w_1,\ldots,w_d \in \mathbf{R}} \sum_{i=1}^{d} (nw_i^2 - 2y_i^T X_i w_i + \lambda w_i^2) + y^T \tag{1.11}$$

$$\min_{w_1,\ldots,w_d \in \mathbf{R}} \sum_{i=1}^{d} ((n+\lambda) w_i^2 - 2y_i^T X_i w_i) + y^T y \tag{1.12}$$

Setting the gradient to zero:

$$\frac{\partial L}{\partial w_i} = 2(n+\lambda) w_i - 2y^T X_i = 0 \tag{1.13}$$

$$w_i^* = \frac{y^T X_i}{n+\lambda} \tag{1.14}$$

Here, we can see that if $w_i = 0 \rightarrow y^T X_i = 0$ and is independent on the threshold lambda.

2 Ridge Regression Classifier Vs. SVM

- a: Given data $X \in \mathbb{R}^{m,n}$ and labels $y \in \{0,1\}^m$, we could create a ridge regression classifier as follows:
  1. Let $\phi(y_i) = \{-1 : y_i = 0, 1 : y_i = 1\}$
  2. let $y_i' = \phi(y_i) : \forall\, y_i$
  3. retrieve w^*_{ridge} on data X and labels y'
  4. and let the classifier function f$(x) = \{-1 : x^T w^* \leq 0 : 1 : \text{otherwise}\}$
- b: this portion is on the iPython Notebooks.

3 PCA REVISITED WITH GRADIENT DESCENT

- a: the function $x^T Cx$ is convex, $-x^T Cx$ is a concave function and the set of x, which is not convex.that are feasible for the problem specified will remain the same after we find the optimal value and negate it (it is just an affine transformation). Therefore, the problem is not convex. (go over this in OH)
  We know the maximizer of the function is the eigenvector corresponding to the maximum eigenvalue. The maximum value of this function is $\lambda_{max} C$
  Consider solving the dual,

$$-\mathscr{L}(x,\lambda) = x^T Cx - \lambda(\|x\|_2^2 - 1) \tag{3.1}$$

$$g(\lambda) = \min_x \mathscr{L}(x,\lambda) \tag{3.2}$$

$$\nabla_x \mathscr{L} = (-C + \lambda I)x = 0 \tag{3.3}$$

$$\min_x \mathscr{L}(x,\lambda) = \begin{cases} -\infty & -Cx + \lambda x \neq 0 \\ \lambda & \text{else} \end{cases}$$

  Our dual formulation is the following:

$$\max \lambda$$

$$\text{s.t} : (-C + \lambda I)x = 0$$

  Where the value of the function is the maximum eigenvalue of C.

- b: Let $f(x) = x^T Cx, g(x) = x^T x$

$$(f/g)(x)' = \frac{f'g - fg'}{g^2} \tag{3.4}$$

$$= \frac{\|x\|_2^2 Cx - (x^T Cx) I_d x}{\|x\|_2^4} \tag{3.5}$$

- c:

$$\lim_{t\to\infty} \frac{Y_{t-1}^T C Y_{t-1}}{Y_{t-1}^T Y_{t-1}} = \frac{Y_{t-1}^T \sum_t X_t X_t^T Y_{t-1}}{t Y_{t-1}^T Y_{t-1}} \tag{3.6}$$

$$= \frac{1}{t}\sum_{i=1}^{t} f_i(Y_{t-1}) \tag{3.7}$$

$$\text{Where } f_i(y_{t-1}) = \frac{y_{t-1}^T X_t X_t^T y_{t-1}}{\|y_{t-1}\|_2^2}$$

$$\nabla f_k(Y_{t-1}) = \frac{\|Y_{t-1}\|_2^2 X_t X_t^T Y_{t-1} - (Y_{t-1}^T X_t X_t^T Y_{Id}) Y_{t-1}}{\|Y_{t-1}\|_2^4} \tag{3.8}$$

$$\nabla f_k(Y_{t-1}) = \frac{\|Y_{t-1}\|_2^2 X_t X_t^T - (Y_{t-1}^T X_t X_t^T I_d Y_{t-1})}{\|Y_{t-1}\|_2^4} Y_{t-1} \tag{3.9}$$

$$= \left(\frac{X_t X_t^T}{\|Y_{t-1}\|_2^2} - \frac{Y_{t-1}^T X_t X_t^T Y_{t-1} I_d}{\|Y_{t-1}\|_2^4}\right) Y_{t-1} \tag{3.10}$$

Which implies the following gradient update rule.

$$Y_t = Y_{t-1} + \gamma_t \left(\frac{X_t X_t^T}{\| Y_{t-1} \|_2^2} - \frac{Y_{t-1}^T X_t X_t^T Y_{t-1} I_d}{\| Y_{t-1} \|_2^4} \right) Y_{t-1}$$

Which is equivalent to the following if we absorb the norm squared term into the step-size.

$$Y_t = Y_{t-1} + \gamma_t \left(X_t X_t^T - \frac{Y_{t-1}^T X_t X_t^T Y_{t-1} I_d}{\| Y_{t-1} \|_2^2} \right) Y_{t-1}$$

- this portion is in the iPython Notebook.

4 Sparse Probability Optimization

- Let $u \in \{0,1\}^n$ s.t $\mathbb{1}^T u \leq k$

$$u^T f^{\min} + (1-u)^T f^0 \tag{4.1}$$
$$\geq \min_u u^T f^{\min} + (1-u)^T f^0 \tag{4.2}$$
$$\forall u : u^T f^{\min} + (1-u)^T f^0 \geq \min_u u^T f^{\min} + (1-u)^T f^0 \tag{4.3}$$
$$p^* \geq q^* \tag{4.4}$$

- b: Let u^* be a minimizer of q^*, because we know this means $u^T x \leq k$
  Let $x' = \begin{bmatrix} u_1 x_1 \\ u_2 x_2 \\ . \\ . \\ . \\ u_n x_n \end{bmatrix}$, because we $u^T x \leq k \rightarrow \text{card}(x') \leq k$ therefore, we know x' is a feasible point in problem 1.

$$\sum_{i=1}^{n} f_i(x_i') \geq \min_x \sum_{i=1}^{n} f_i(x_i)$$
$$q^* \geq p^*$$

- c:

$$q^* = \min_u u^T f^{\min} + (\mathbb{1} - u)^T f^0 \tag{4.5}$$
$$= u^T f^{\min} + \mathbb{1}^T f^0 - u^T f^0 \tag{4.6}$$
$$= u^T (f^{\min} - f^0) + \mathbb{1}^T f^0 \tag{4.7}$$
$$= \min_u \mathbb{1}^T f^0 - u^T (f^0 - f^{\min}) \tag{4.8}$$
$$= \mathbb{1}^T f^0 - s_k (f^0 - f^{\min}) \tag{4.9}$$

5 HYDRO-ELECTRIC POWER GENERATION

- a: In general, we know the volumes of the reservoirs must meet a flow equation:

$$V_{l,k} = V_{l,k-1} + A_{l,k} + \sum_{j \in U_i} T_{j,k} - \sum_{j \in D_t} T_{j,k}$$

We know that reservoirs one and two both receive water inflow $A_{2,i}$ and $A_{2,i}$ respectively at time i. We also know that the first reservoir does not have any incoming flows from turbines, but has two out going flows from turbines one and two. This gives us the following equations for the first reservoir.

$$V_{1,i} = V_{1,i-1} + A_{1,i} - (T_{1,i} + T_{2,i})$$

Those turbines that flow out of the first reservoir flow into the second reservoir and there is a third turbine that flows out of the second reservoir which gives us the following equations.

$$V_{2,i} = V_{2,i-1} + A_{2,i} + (T_{1,i} + T_{2,i}) - T_{3,i}$$

Therefore, the optimization problem given is now as follows:

$$\min_{v,t} \sum_{l=1}^{L} w_l V_{l,0} - (\sum_{l=1}^{L} w_l V_{l,k} + \sum_{k=1}^{K} \sum_{j=1}^{J} \lambda_k \rho_j T_{j,k})$$

$$\text{s.t } T_{\min} \leq T \leq T_{\max}$$

$$V_{\min} \leq V \leq V_{\max}$$

$$V_{1,i} = V_{1,i-1} + A_{1,i} - (T_{1,i} + T_{2,i}) : \forall i \in \{1,2,3\}$$

$$V_{2,i} = V_{2,i-1} + A_{2,i} + (T_{1,i} + T_{2,i}) - T_{3,i} : \forall i \in \{1,2,3\}$$

- b: This portion is in the iPython Notebook.
- c: To limit the number of turbines that would be affected by a change, we can go ahead and punish the l_1 norm of the difference of T and $\hat{T}$. The encouraged sparsity of the difference of the two matrices will correspond to the values in the matrix T being kept the same.
- d: this portion is in the iPython Notebook.