Statistical Machine Learning - Exercise 4

Dominik Marino - 2468378, Pertami Kunz - 2380210
August 8, 2020

Contents

1 Neural Network

Given the mnist dataset that contains $N_{tr} = 6006$ training and $N_{te} = 1004$ test data points. Each input has a dimension of $784 = 28 \times 28$, corresponding to a flattened image dataset. One example of such dataset is shown below.

Training digit:0

```python
import numpy as np
import matplotlib.pyplot as plt

# Training Dataset
Xtrain = np.genfromtxt("mnist_small_train_in.txt", delimiter=',', dtype=int)
ytrain = np.genfromtxt("mnist_small_train_out.txt", dtype=int)

# Test Dataset
Xtest = np.genfromtxt("mnist_small_test_in.txt", delimiter=',', dtype=int)
ytest = np.genfromtxt("mnist_small_test_out.txt", dtype=int)

# Check 1 training sample
plt.imshow(Xtrain[0,:].reshape((28,28)))
plt.title('Training digit:' + str(ytrain[0]))
plt.axis("off")
plt.show
```

Listing 1: Reading data and displaying one training sample.

1.1 Multi-layer Perceptron

We started by exploring this selected architecture: 2-layer neural network (1 hidden layer), with 300 nodes in the hidden layer, because it is the simplest network that should give the accuracy of at least 95%/ error of at most 5% (on the full dataset)[1].

Formulation

With 1 hidden layer (2-layer NN),

$$y_k(\mathbf{x}) = f^{(2)} \left(\underbrace{W_{k,0}^{(2)} + \underbrace{\sum_{i=1}^{h} W_{k,i}^{(2)} f^{(1)} \left(\underbrace{\sum_{j=0}^{d} W_{i,j}^{(1)} \underbrace{x_j}_{a^{(0)}}}_{z^{(1)}} \right)}_{a^{(1)}}}_{z^{(2)}} \right) \tag{1}$$

where $f^{(1)}(\cdot)$ and $f^{(2)}(\cdot)$ are activation functions in the hidden and output layers, respectively, and, in this case,

- $k = 0, \ldots, 9$
- $i = 1, \ldots, 300$ or $h = 300$
- $j = 0, \ldots 784$ or $d = 784$

[1] http://yann.lecun.com/exdb/mnist/

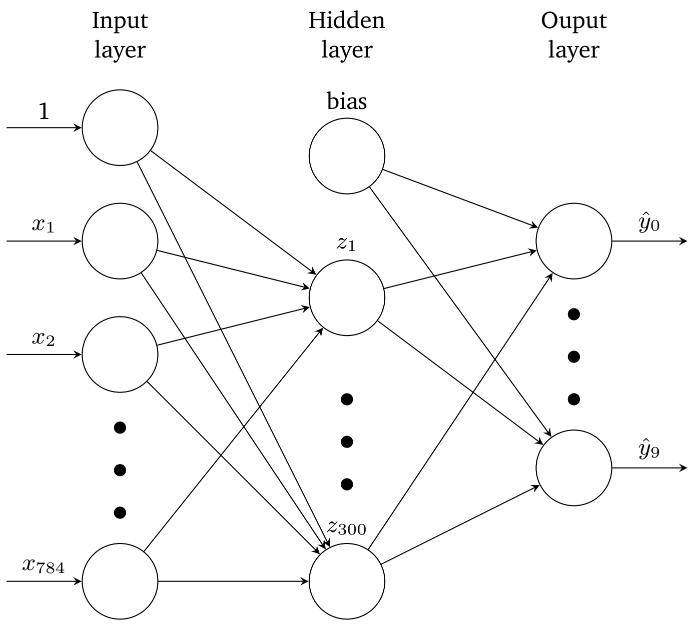


Figure 1: Selected NN architecture for the MNIST dataset.

- $x_0 = 1$ and $W^{(1)}_{i,0}, W^{(2)}_{k,0}, \ \forall i, k$ correspond to the bias.

<u>**Activation functions**</u>
We tried using **tanh** and **sigmoid** functions for the hidden layer and **softmax** for the output layer:

$$f^{(1)}(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$
$$\text{or } f^{(1)}(z) = \tanh(z) = \frac{2}{1+e^{-2z}} - 1$$
$$f^{(2)}_k(\mathbf{z}) = \hat{y}_k = \frac{e^{z_k}}{\sum_{l=0}^{9} e^{z_l}}.$$

The derivative of the sigmoid function:

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = -\frac{1}{(1+e^{-z})^2} \cdot \frac{d}{dz}\left(1+e^{-z}\right) = \frac{e^{-z}}{(1+e^{-z})^2} = \ldots = \sigma(z)\left(1-\sigma(z)\right). \tag{2}$$

The derivative of the tanh function:

$$f'^{(1)}(z) = \frac{d}{dz}\tanh(z) = 1 - \tanh^2(z) \tag{3}$$

<u>**Loss function**</u>
Given the true label y_k and using 1-hot encoder,

$$y_k = \begin{cases} 1 & \text{if } y = k \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

For classification, we use cross-entropy loss function:
Cross entropy loss function

$$L = -\sum_{k=0}^{9} y_k \ln(\hat{y}_k) \tag{5}$$

and since only one y_k has the value of 1, the derivative of cross entropy loss function with softmax activation function:

$$L = -\ln(\hat{y}_k) = -\ln\left(\frac{e^{z_k}}{\sum_{l=0}^{9} e^{z_l}}\right) = -z_k + \ln\sum_{l=0}^{9} e^{z_l}. \tag{6}$$

The derivative for each z_k:

$$\begin{aligned}\nabla_{z_i} L = \nabla_{z_i}\left(-z_k + \ln \sum_{l=0}^{9} e^{z_l}\right) &= \nabla_{z_i} \ln \sum_{l=0}^{9} e^{z_l} - \nabla_{z_i} z_k \\ &= \frac{1}{\sum_{l=0}^{9} e^{z_l}} \nabla_{z_i} \sum_{l=0}^{9} e^{z_l} - \nabla_{z_i} z_k \\ &= \frac{z_i}{\sum_{l=0}^{9} e^{z_l}} - \nabla_{z_i} z_k \\ &= \hat{y}_i - \delta(i) \\ &= \hat{y}_i - y_i. \end{aligned} \tag{7}$$

Forward- propagation

- $\mathbf{a}^{(0)} = \mathbf{x}$ (already includes bias: $\mathbf{x} = [1, x_1, \ldots, x_{784}]^\top$)
- $\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}$
  - $\mathbf{z}^{(1)} = [z_1, \ldots, z_{300}]^\top$, where $z_i = \mathbf{w}_i^\top \mathbf{x}$ and $\mathbf{w}_i = \left[W_{i,0}^{(1)}, W_{i,1}^{(1)}, \ldots, W_{i,784}^{(1)}\right]$, for $i = 1, \ldots, 300$.
  - Matrix–vector form:

$$\mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ \vdots \\ z_{300}^{(1)} \end{bmatrix} = \begin{bmatrix} W_{1,0}^{(1)} & W_{1,1}^{(1)} & \cdots & W_{1,784}^{(1)} \\ W_{2,0}^{(1)} & W_{2,1}^{(1)} & \cdots & W_{2,784}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{300,0}^{(1)} & W_{300,1}^{(1)} & \cdots & W_{300,784}^{(1)} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_{784} \end{bmatrix}.$$

- $\mathbf{a}^{(1)} = f^{(1)}\left(\mathbf{z}^{(1)}\right) = \sigma\left(\mathbf{z}^{(1)}\right)$
- $\mathbf{z}^{(2)} = \mathbf{b} + \mathbf{W}^{(2)}\mathbf{a}^{(1)}$
- Matrix–vector form:

$$\mathbf{z}^{(2)} = \begin{bmatrix} z_0^{(2)} \\ z_1^{(2)} \\ \vdots \\ z_9^{(2)} \end{bmatrix} = \begin{bmatrix} W_{0,0}^{(2)} \\ W_{1,0}^{(2)} \\ \vdots \\ W_{9,0}^{(2)} \end{bmatrix} + \begin{bmatrix} W_{0,1}^{(2)} & \cdots & W_{0,300}^{(2)} \\ W_{1,1}^{(2)} & \cdots & W_{1,300}^{(2)} \\ \vdots & \ddots & \vdots \\ W_{9,1}^{(2)} & \cdots & W_{9,300}^{(2)} \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_{300} \end{bmatrix}.$$

- $\hat{y}_k = f_k^{(2)}\left(\mathbf{z}^{(2)}\right) = \frac{e^{z_k}}{\sum_{l=0}^{9} e^{z_l}}$, for $k = 0, \ldots, 9$.

Backward-propagation

The update rule: $\mathbf{W}^{(\ell)} = \mathbf{W}^{(\ell)} - \alpha \frac{\partial L}{\partial \mathbf{W}^{(\ell)}}$, where α is the learning rate.

- $\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{a}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}}$
- $\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{z}^{(2)}}$
- $\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{a}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}}$.
  - From (7): $\frac{\partial L}{\partial \mathbf{z}^{(2)}} = \hat{\mathbf{y}} - \mathbf{y}|_{10\times 1}$
  - $\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}} = \frac{\partial}{\partial \mathbf{W}^{(2)}}\left(\mathbf{b} + \mathbf{W}^{(2)}\mathbf{a}^{(1)}\right) = \mathbf{a}^{(1)\top}\Big|_{1\times 300}$
  - $\frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} = \frac{\partial}{\partial \mathbf{a}^{(1)}}\left(\mathbf{b} + \mathbf{W}^{(2)}\mathbf{a}^{(1)}\right) = \mathbf{W}^{(2)}\Big|_{300\times 10}$
  - From (2) and (3): $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} = \sigma'\left(z^{(1)}\right)$ or $\tanh'\left(z^{(1)}\right)\Big|_{300\times 1}$
  - $\frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} = \mathbf{x}^\top\Big|_{1\times 785}$.

Then the gradients

- $\frac{\partial L}{\partial \mathbf{W}^{(2)}}\Big|_{10\times 300} = \frac{\partial L}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}} = \hat{\mathbf{y}} - \mathbf{y}\big|_{10\times 1} \cdot \mathbf{a}^{(1)\top}\Big|_{1\times 300}$
- With dimension corrections,

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(1)}}\Big|_{300\times 785} &= \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{a}^{(1)}} \frac{\partial L}{\partial \mathbf{z}^{(2)}} \otimes \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \cdot \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} \\ &= \left(\underbrace{\mathbf{W}^{(2)\top}}_{300\times 10} \underbrace{(\hat{\mathbf{y}} - \mathbf{y})}_{10\times 1} \right) \otimes \underbrace{\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}}}_{300\times 1} \cdot \underbrace{\mathbf{x}^{\top}}_{1\times 785},\end{aligned}$$

  where $\otimes$ is element-wise multiplication.

Preprocessing

$x_0 = 1$ is added to each sample in the training and test set. The y-values are given in all possible numbers $0, 1, \ldots, 9$. We first convert this into one-hot-encoding, such that

$$0 \to [1,0,0,0,0,0,0,0,0,0] \quad \cdots \quad 9 \to [0,0,0,0,0,0,0,0,0,1]$$

Initialization

For the weight matrices, Xavier's initialization is used.[2]

```
## PREPROCESSING
Ntr = ytrain.shape[0] # No. of training samples
Nte = ytest.shape[0] # No. of training samples
K = 10 # No. of classes
L1 = 785 # No. of nodes in input layer
L2 = 300 # No. of nodes in hidden layer

# Add x0=1
Xtrain = np.hstack((np.ones((Ntr,1)), Xtrain))
Xtest = np.hstack((np.ones((Nte,1)), Xtest))
# One-Hot-Encoding
Ytrain = np.eye(K)[ytrain]
Ytest = np.eye(K)[ytest]

## Initialization
np.random.seed(1)
# weight matrices and bias
W1 = np.random.rand(L2,L1) / np.sqrt(L1)
W2 = np.random.rand(K,L2) / np.sqrt(L2)
bias = np.random.rand(K,1) / np.sqrt(K)
```

Listing 2: Preprocessing and initialization.

Training and Testing

The weight updates are done in mini batches (averaged per mini batch). For each epoch run, the training samples are first randomized. The different learning rates as seen in the results in Figure 2 were chosen simply because they seem to work, we tried different values but they failed to improve the training errors after many epochs/very slow.

With minibatch size of 256 and learning rates of 1 and 0.1 respectively for first->hidden and hidden->output layers, we obtained the best performance after about 1000 epochs: testing error of $66/1004 \times 100\% \approx 6.6\%$. Beyond 1000 epochs, the network started to be overfitting to the training data, as seen in Figure 3, where training error keeps declining while the testing error starts increasing.

With only one fully-connected (dense) hidden layer, it takes many epochs to train the network to get <8% test error. This should be faster if convolutional NN or more hidden layers were to be used.

```
#############################################################################
def tanh_grad(a):
    return 1-(np.tanh(a))**2
def sigmoid(a):
    return np.where(a>= 0, 1/(1 + np.exp(-a)), np.exp(a)/(1 + np.exp(a)))
def sig_grad(a):
    return sigmoid(a) * (1 - sigmoid(a))
def softmax(a):
    e_a = np.exp(a - np.max(a)) # max(a) for numerical stability
    return e_a / e_a.sum(axis=0)
def Loss(y, y_hat):
    return np.sum(y * np.log(y_hat + 1e-6))  # add 1e-6 for numerical stability

## TRAINING & TESTING
## =>> WARNING!! SLOW!! NOT OPTIMIZED WHATSOVER, Works though!
epochs = 20000 # How many complete passes through the training set
```

[2]https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

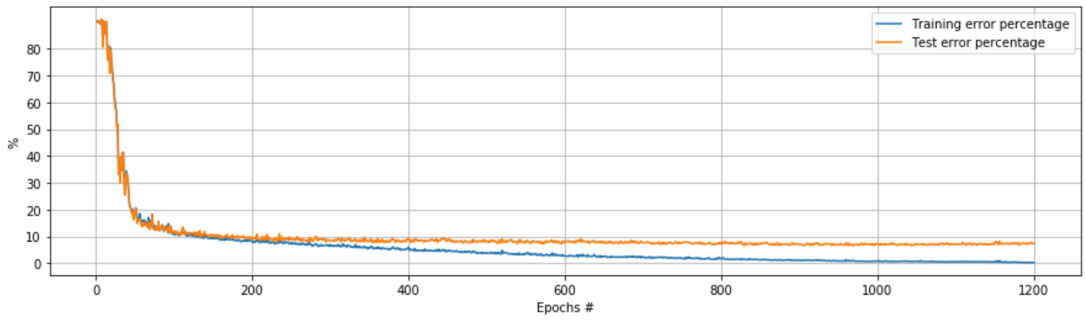


Figure 2: With learning rates of 1 and 0.1 for first->hidden and hidden->output layers, respectively

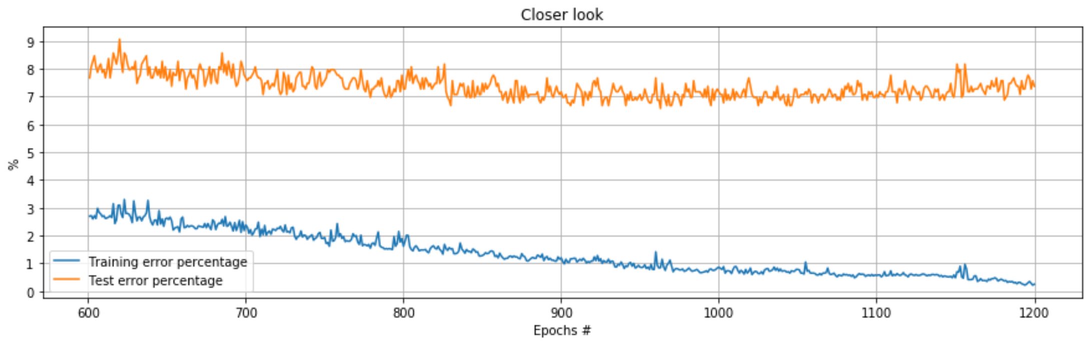


Figure 3: Closer look: overfitting beyond 1000 epochs.

```

batchsize = 256 # just because
alpha1 = 1 # Learning rate
alpha2 = 0.1 # Learning rate
dL_dW1 = 0
dL_dW2 = 0
i=0
train_loss = []
TR = []
TE = []
while i<epochs:
    idx = np.random.permutation(Ntr)
    X = Xtrain[idx,:]
    Y = Ytrain[idx,:]

    # Train with mini batches
    for start_idx in range(0, Ntr, batchsize):
        end_idx = min(start_idx+batchsize, Ntr)

        # mini batch
        Xb = np.transpose(X[start_idx:end_idx,:])
        Yb = np.transpose(Y[start_idx:end_idx,:])

        minibatch_grads1 = np.empty(W1.shape)
        minibatch_grads2 = np.empty(W2.shape)
        minibatch_gradsb = np.empty((K,1))
        minibatch_loss = np.empty(Xb.shape[1])
        for b in np.arange(Xb.shape[1]):
            # Forward feed
            # input => hidden layer
            Z1 = np.dot(W1, Xb[:,b])[:, np.newaxis]
            #a1 = sigmoid(Z1)
            a1 = np.tanh(Z1)
            # hidden layer => output
            Z2 = bias + np.matmul(W2,a1)
            Y_hat = softmax(Z2)

            # Backward feed/ Gradients
            dL_dz2 = (Y_hat - Yb[:,b][:, np.newaxis])
            #da1_dz1 = sig_grad(Z1)
```

```python
            da1_dz1 = tanh_grad(Z1)
            dL_dW2 = np.matmul( dL_dz2 , a1.T )
            dL_dW1 = np.matmul( (np.matmul(W2.T, dL_dz2) * da1_dz1), np.transpose(Xb[:,b][:, np.newaxis]))

            minibatch_grads1 += dL_dW1
            minibatch_grads2 += dL_dW2
            minibatch_gradsb += dL_dz2
            #minibatch_loss[b] = Loss(Yb[:,b], Y_hat)

    # Update
    dW1 = minibatch_grads1 / Xb.shape[1]
    dW2 = minibatch_grads2 / Xb.shape[1]
    db = minibatch_gradsb / Xb.shape[1]
    #train_loss.append(np.mean(minibatch_loss))
    W1 = W1 - alpha1 * dW1
    W2 = W2 - alpha2 * dW2
    bias = bias - alpha2 * db

    # Training error
    tr_e = 0
    for n in range(Ntr):
        x = Xtrain[n,:].T
        Z1 = np.dot(W1, x)[:, np.newaxis]
        #a1 = sigmoid(Z1)
        a1 = np.tanh(Z1)
        Z2 = bias + np.matmul(W2,a1)
        Y_hat = softmax(Z2)
        if np.argmax(Y_hat)!=ytrain[n]:
            tr_e += 1

    # Test error
    te_e = 0
    for n in range(Nte):
        x = Xtest[n,:].T
        Z1 = np.dot(W1, x)[:, np.newaxis]
        #a1 = sigmoid(Z1)
        a1 = np.tanh(Z1)
        Z2 = bias + np.matmul(W2,a1)
        Y_hat = softmax(Z2)
        if np.argmax(Y_hat)!=ytest[n]:
            te_e += 1

    print("epoch #"+str(i+1)+" Tr err: "+str(tr_e) +", Te err: "+str(te_e))
    TR.append(tr_e)
    TE.append(te_e)
    i+=1
```

Listing 3: Training and Testing.

```python
TR_perc = np.array(TR)/Ntr*100
TE_perc = np.array(TE)/Nte*100
plt.figure(figsize=(15,4))
plt.plot(np.arange(1,TR_perc.shape[0]+1), TR_perc, label="Training error percentage")
plt.plot(np.arange(1,TE_perc.shape[0]+1), TE_perc, label="Test error percentage")
plt.yticks(np.arange(0,90,10))
plt.xlabel("Epochs #")
plt.ylabel("%")
plt.legend()
plt.grid('on')
plt.show()

plt.figure(figsize=(15,4))
plt.plot(np.arange(601,TR_perc.shape[0]+1), TR_perc[600:], label="Training error percentage")
plt.plot(np.arange(601,TE_perc.shape[0]+1), TE_perc[600:], label="Test error percentage")
plt.yticks(np.arange(0,10,1))
plt.xlabel("Epochs #")
plt.ylabel("%")
plt.legend()
plt.title("Closer look")
plt.grid('on')
plt.show()
```

Listing 4: Plotting the percentages.

1.2 Deep learning

- **Highlight the qualitative differences between classical neural networks and deep networks.**
  While both classical/shallow neural networks and deep networks try to imitate how human nerve cells work, deep networks usually involve many more hidden layers to uncover some patterns/relationships in the features. The architecture of deep networks is more complex, but the concepts stay the same. In a deep network, the function value y is a composition of many-level function
  $$y = (f_K \circ f_{K-1} \circ \ldots \circ f_1)(x) \,,$$
  where K is typically >3, x are the inputs (e.g. images), and y are the observations (e.g. class labels). By adding more layers and nodes, a deep network can represent more complex functions and learn more features compared to a shallow network.

- **Which limitations of classical NN does deep learning overcome? Give an intuition of the innovations introduced in deep learning compared to traditional NN.**
  In classical NN, one may need to perform labor-intensive feature-engineering. In deeper NN, the network depends less on feature engineering; it will extract and organize the discriminative information from the data. Deep learning methods showed promise in using the data itself to learn new good features and have been very successful in signal processing, computer vision, speech recognition, and natural language processing.
  Deep networks promote the re-use of features, and can potentially lead to more abstract features. Deep learning enables representation learning, which can be convenient to express general priors (smoothness, sparsity, temporal and spatial coherence, etc) about the world which would be likely to be useful to solve AI-tasks.
  In deeper networks, one can forego using fully connected (dense) layers. Dense layer may be prohibitive in terms of computation time without improving the accuracy of the model. In deep networks, one can add convolutional/recurrent networks in the layers closer to the input which normally have much a lot more nodes. The fewer connections help improving the performance. This is made possible through convolutional and recurrent networks for taking into account spatial and temporal relationships among the inputs. With deeper networks it is also possible to incorporate regularization techniques like Dropout to randomly "kill" some nodes to avoid overfitting.

- **Are deeper networks necessarily better from a theoretical perspective?**
  In a shallow network you may need to train the weights a lot more times to reach the accuracy (or even lower accuracy) of a deeper network. However, a deeper network also has high computational cost due to a lot more connections which means a lot of weight to update. Due to the millions of parameters, getting enough labeled data to train is difficult. Furthermore, a deeper network may fit to the data on hand too well, i.e. overfitting. Therefore, it is not necessarily better.

  [3] [4] [5] [6]

[3] Goodfellow, I., Bengio, Y. and Courville, A., 2016. Deep learning. MIT press.

[4] Murphy, Kevin P. Machine learning: a probabilistic perspective. MIT press, 2012.

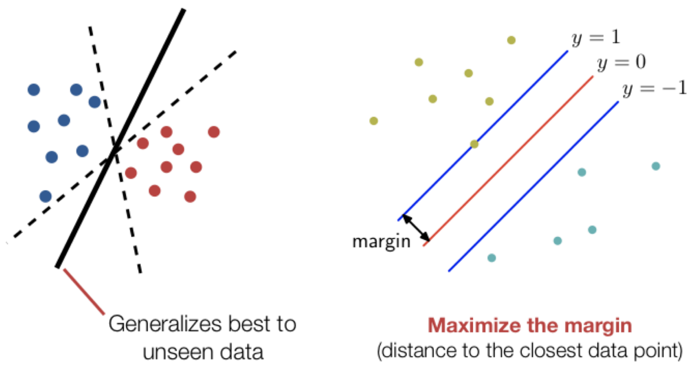
[5] Bengio, Yoshua, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives." IEEE transactions on pattern analysis and machine intelligence 35.8 (2013): 1798-1828.

[6] Deisenroth, A. Aldo Faisal Marc Peter, and Cheng Soon Ong. "Mathematics for Machine Learning. 2019." url: https://mml-book. github. io: 407.

2 Support Vector Machines

2.1 Definition

- **Definition of SVM**
  - Support Vector Machine is a supervised machine learning algorithm which can classification and regression challenges. SVM is a margin classifier and supports linear and non-linear classification. With the SVM algorithm, we can plot each data in a n-dimensional space (n is the number of features) and then perform by finding the best hyperplane do seperate two or more classes. SMV tries to find the best margin (distance between the line and the support vector) that seperates the classes and reduce the risk of an error in the data[7].


$y = 1$
$y = 0$
$y = -1$
margin
Generalizes best to unseen data
Maximize the margin
(distance to the closest data point)

- **Advantages with respect to (w.r.t) other linear approches**
  - SVM is more effective in high dimensional space. Meanwhile in regression the gram matrix X^TX is not invertible. The linear model $\hat{\beta} = (X^TX)^{-1}X^Ty$ completly fails. To improve this model, we can use ridge regression with additional penalty.

$$\hat{\beta}^{Rigde} = argmin_{\beta \in R}||Y - X||_2^2 + \lambda||\beta||_2^2$$

    Logistic regresssion focus on maximinzing the probability of the data and SVM tries to find the seperating hyperplane that maximizes the distance of the closest points to the margin. SVM dont penalize examples for which the correct decision is made with sufficient confidence. This may SVMs be good for generalization and a robust method against outliers.

    The biggest advantages of using Support Vector Machine over a linear approach is to classify non-linear datasets with non-linear kernels. SVMs depends on their support vectors and not on the entire dataset for solving the decision boundary.

2.2 Quadratic Programming

Constrained optimization problem:

$$\begin{aligned} argmin_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ s.t. \quad & y_i(\mathbf{w}^T x_i + b) - 1 \geq 0 \qquad \forall i \end{aligned}$$

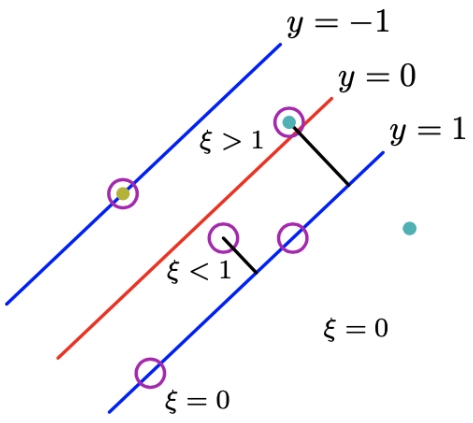
Langrangian Formulation:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i (y_i(\mathbf{w}^T x_i + b) - 1))$$

[7]Picture Source: SML Lecture Script 11 Slide 8

2.3 Slack Variables

- **Explain concept of slack variables**
  - In a linear programming problem is a slack variable the inequality represent the constraint that can be replaced by the equations. The slack variable cannot take negative values. It is always required to be positive or zero. If the slack variable is zero for the data-point, then there are correctly classified and are either on the margin or on the correct side of the margin. If the points $0 < \epsilon < 1$ there lie inside the margin, but on the correct side of the decision boundary. If the data-point for the $\epsilon > 0$ they lie on the wrong side of the decision boundary and they are missclassified. Our goal is to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary. Shown in the grafhic[8].



$$\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\epsilon_i$$

Parameters:

  * C: Costparameter for the missinformation. It is a regularization coefficient because it contrails the trade-off beween minimizing training errors and controlling model complexity.
  * ϵ: non-negative slack variable $\longrightarrow \sum_n \epsilon_n$: is a upperbound on the number of misclassified points
  * $\mathbf{w}$: targetfunction
  * b: bias term

- Reformulate the optimizaiton problem

$$\begin{aligned} argmin_{\mathbf{w},b,\epsilon} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\epsilon_i \\ s.t. \quad & y_i(\mathbf{w}^T x_i + b) \geq 1 - \epsilon_i \\ s.t. \quad & \epsilon_i \geq 0 \end{aligned}$$

Langrangian Formulation:

$$L(\mathbf{w}, b, \alpha, \beta) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\epsilon_i - \sum_{i=1}^{N}\alpha_i(y_i(\mathbf{w}^T x_i + b) - 1 + \epsilon_i)) - \sum_{i=1}^{N}\beta_i\epsilon_i$$

with $\alpha_i > 0, \beta_i > 0$

[8]Picture Source: Pattern Recognition and Machine Learning page 332

2.4 Slack Variables - Solving Dual optimization problem

Langrangian Formulation:

$$\begin{aligned} L(\mathbf{w}, b, \alpha, \beta) &= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\epsilon_i - \sum_{i=1}^{N}\alpha_i(y_i(\mathbf{w}^T x_i + b) - 1 + \epsilon_i)) - \sum_{i=1}^{N}\beta_i\epsilon_i \\ \text{with } &\alpha_i > 0, \beta_i > 0 \end{aligned}$$

Set deviate of L w.r.t w and b to zero:

$$\begin{aligned} \nabla_w L \overset{!}{=} 0 \Rightarrow & w - \sum_{i=1}^{N}\alpha_i\beta_i x_i \overset{!}{=} 0 \\ \Rightarrow & w = \sum_{i=1}^{N}\alpha_i\beta_i x_i \end{aligned}$$

$$\begin{aligned} \nabla_b L \overset{!}{=} 0 \Rightarrow & -\sum_{i=1}^{N}(\alpha_i\beta_i w^T x_i + \alpha_i y_i b - \alpha_i + \epsilon_i\alpha_i) \overset{!}{=} 0 \\ \Rightarrow & -\sum_{i=1}^{N}\alpha_i y_i = 0 \\ \Rightarrow & \sum_{i=1}^{N}\alpha_i y_i = 0 \end{aligned}$$

$$\begin{aligned} \nabla_\epsilon L \overset{!}{=} 0 \Rightarrow & C\sum_{i=1}^{N}1 - \sum_{i=1}^{N}\beta_i - \sum_{i=1}^{N}\alpha_i \\ \Rightarrow & C * N - N * \beta_i - N * \alpha_i = 0 \\ \Rightarrow & \alpha_i = C - \beta_i \end{aligned}$$

$$\nabla_\alpha L \overset{!}{=} 0 \Rightarrow \sum_{i=1}^{N} y_i(w^T x_i + b) - 1 + \epsilon_i) = 0$$

$$\nabla_\beta L \overset{!}{=} 0 \Rightarrow \sum_{i=1}^{N}\epsilon_i = 0$$

Substituting in Lagrangian:

$$\begin{aligned} L(\mathbf{w}, b, \alpha, \beta) &= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\epsilon_i - \sum_{i=1}^{N}\alpha_i(y_i(\mathbf{w}^T x_i + b) - 1 + \epsilon_i)) - \sum_{i=1}^{N}\beta_i\epsilon_i \\ &= \frac{1}{2}w^T w + C\sum_{i=1}^{N}\epsilon_i - \sum_{i=1}^{N}\alpha_i y_i w^T x_i - \sum_{i=1}^{N}\alpha_i y_i b + \sum_{i=1}^{N}\alpha_i - \sum_{i=1}^{N}\alpha_i\epsilon_i - \sum_{i=1}^{N}\beta_i\epsilon_i \end{aligned}$$

With the derivation we can simplify the terms:

$$\begin{aligned}
L(\mathbf{w},b,\alpha,\beta) &= \frac{1}{2}w^Tw + C\cancelto{0}{\sum_{i=1}^{N}}\epsilon_i - \sum_{i=1}^{N}\alpha_i y_i w^T x_i - \cancelto{0}{\sum_{i=1}^{N}}\alpha_i y_i b + \sum_{i=1}^{N}\alpha_i - \sum_{i=1}^{N}\alpha_i\epsilon_i - \sum_{i=1}^{N}\beta_i\epsilon_i \\
&= \sum_{i=1}^{N}\alpha_i + \frac{1}{2}w^Tw - \sum_{i=1}^{N}\alpha_i y_i w^T x_i - \sum_{i=1}^{N}\alpha_i\epsilon_i - \sum_{i=1}^{N}\beta_i\epsilon_i \\
&= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - \sum_{i=1}^{N}(\alpha_i\epsilon_i + \beta_i\epsilon_i) \\
&= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - \sum_{i=1}^{N}\epsilon_i(\alpha_i + \beta_i)
\end{aligned}$$

Set $\alpha_i = C - \beta_i$ in:

$$\begin{aligned}
L(\mathbf{w},b,\alpha,\beta) &= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - \sum_{i=1}^{N}\epsilon_i(C - \beta_i + \beta_i) \\
&= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - \sum_{i=1}^{N}\epsilon_i(C) \\
&= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - C\sum_{i=1}^{N}\epsilon_i \\
&= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}w - \sum_{i=1}^{N}\alpha_i y_i x_i] - C\cancelto{0}{\sum_{i=1}^{N}}\epsilon_i
\end{aligned}$$

Now we can set the formula for w = $\sum_{i=1}^{N}\alpha_i\beta_i x_i$ in our Lagrangian from our derivation:

$$\begin{aligned}
L(\mathbf{w},b,\alpha,\beta) &= \sum_{i=1}^{N}\alpha_i + w^T[\frac{1}{2}\sum_{i=1}^{N}\alpha_i\beta_i x_i - \sum_{i=1}^{N}\alpha_i y_i x_i] \\
&= \sum_{i=1}^{N}\alpha_i + w^T[-\frac{1}{2}\sum_{i=1}^{N}\alpha_i\beta_i x_i] \\
&= \sum_{i=1}^{N}\alpha_i + (\sum_{i=1}^{N}\alpha_i\beta_i x_i)^T[-\frac{1}{2}\sum_{i=1}^{N}\alpha_i\beta_i x_i] \\
&= \sum_{i=1}^{N}\alpha_i + \sum_{i=1}^{N}\alpha_i\beta_i x_i^T[-\frac{1}{2}\sum_{i=1}^{N}\alpha_i\beta_i x_i] \\
&= \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=0}^{N}\alpha_i\alpha_j y_i y_j x_i^T x_j
\end{aligned}$$

Parameters:

- Scaler:
  - α: Lagrange multiplier for training example
  - y: -1 or 1
- Vector:
  - x: feature vector of training data

Dual Optimization Problem:

$$\begin{aligned} max_\alpha \quad & W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=0}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j \\ s.t \quad & \alpha_i \geq 0 \qquad i = 1, ..., m \\ & \sum_{i}^{m} \alpha_i y_i = 0 \end{aligned}$$

Hence, we can solve dual in lieu of primal problem:

$$max_\alpha \quad W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j < x_i, x_j >$$

- $< x_i, x_j >$: Inner product of feature vector

If we want to solve the features in higer dimensional space, we have to replace the features x by $\phi(x)$

2.5 The Dual Problem

- **Advantages of solving the dual instead of the primal?**

  Primal:

$$min_{w,b} max_{\alpha \geq 0} \quad \frac{1}{2} ||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i (y_i (\mathbf{w}^T x_i + b) - 1))$$

  Dual:

$$max_{\alpha \geq 0} min_{w,b} \quad \frac{1}{2} ||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i (y_i (\mathbf{w}^T x_i + b) - 1))$$

  - The dual problem allows it to reformulate the model using kernels and maximize the margin classifier that can efficiently applied to the feature space. The most significant benefit form solving it dual is the Kernel Trick (explained in the 2.6). The dual Kernel evaluation requires less kernel evaluation then the primal. This gave the dual a more stable solution in less computation time. The training time of the dual SVM is faster then the primal SVM and usually the number of the support vector are smaller. An another advantages is that the dual problem has many variables as constraints as there are in the primal. So we can reduce the dimensionality of the matrix so much[9].

2.6 Kernel Trick

- **Explain the kernel trick and why is it particularly convenient in SVMs**
  - The Lagrangian trasform the SVM problem to get a good generalization for our data and so we can drop the slack variable. Langrangian allows us to take the constrained optimization problem and reformulate it as an unconstrained problem, like:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2} ||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i (y_i (\mathbf{w}^T x_i + b) - 1))$$

    The minus sign with the second term is used because we are minimizing with respect to the first term, but maximizing the second. Now we can simplifying the result, and making use of our constraint and derive our model:

    Dual Lagrangian:

$$W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=0}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

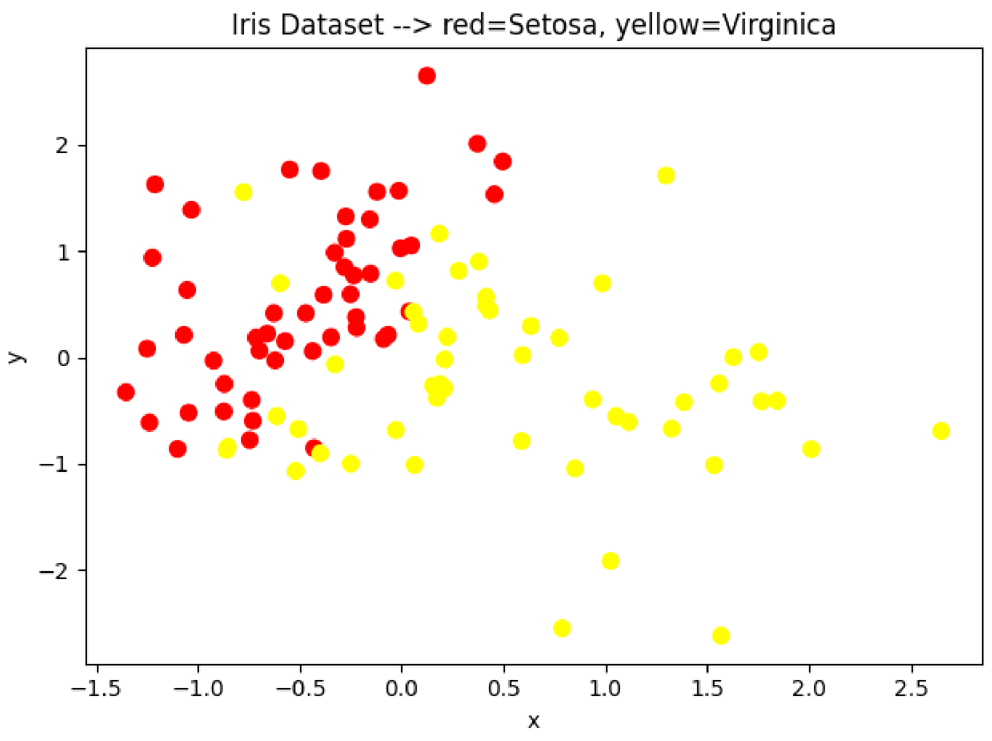
    This objective function is actually more expensive to evaluate than the primal Lagrangian. So it leads to following formula form:

$$W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i} \sum_{j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

[9]Paper from Shigeo Abe: Is primal better then dual `https://www.researchgate.net/publication/262407522_Is_Primal_Better_Than_Dual`

where $k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ is a kernel function. With the kernel trick it frees us from thinking about the features directly. We can classify data that is not linearly separable in the original feature's space. In our feature space we dont have to compute the coordinates of the data in the space, but we can easily compute the inner product of all pairs of data in our feature space, because it is a lot of easier to get inner product in a high dimensional space then the actual points in a high dimensional space. It is computationally cheaper then the explicit computation of the coordinates.

2.7 Implementation



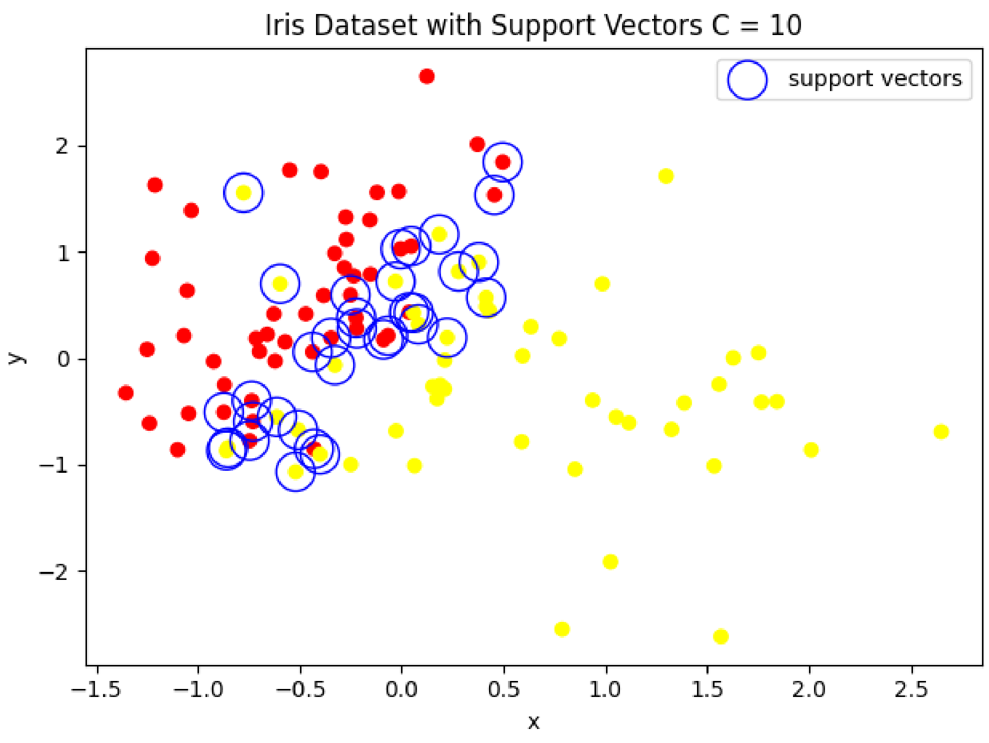
We see in our data, that we can use a linear kernel to solve this task. We also get a more generalization of our data to get a nicer hyperplane between Setosa and Virginica. A polynomial kernel would probably overfitting. For this case a non-linear kernel (polynomial, radial basis funciton, etc.) wouldn't be usefull.

```python
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix as cvxopt_matrix
from cvxopt import solvers as cvxopt_solvers

iris = np.genfromtxt('dataSets/iris-pca.txt')
X_data, y_data = iris[:, 0:2], iris[:, 2]

def show_data(X, y):
    plt.figure()
    plt.title('Iris Dataset --> red=Setosa, yellow=Virginica')
    plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

Listing 5: Iris Dataset

```python
def main():
    def change_labels(labels):
        Y = []
        for y in labels:
            if y == 0:
                Y.append(-1)
            else:
                Y.append(1)
        return np.asarray(Y)
    X, y = X_data, change_labels(y_data)
    show_data(X, y)
    doSMV(X, y, linear_kernel, 10)

if __name__ == '__main__':
    main()
```

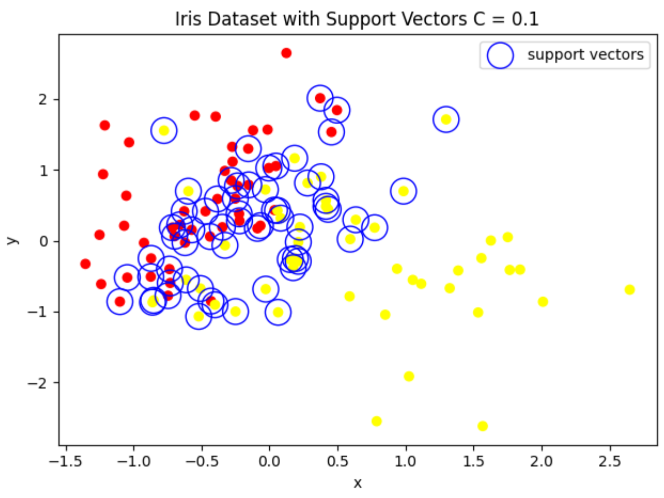
Listing 6: Main

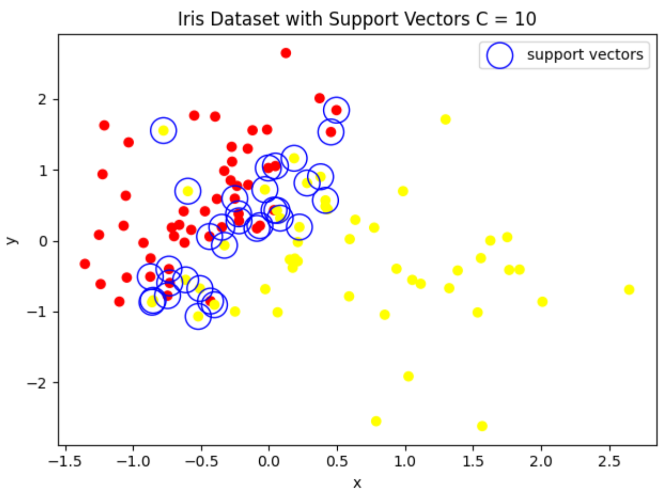


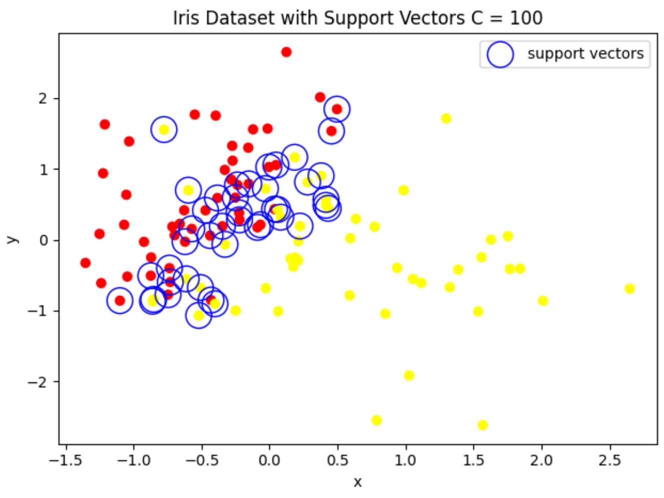
```
def doSMV(X, y, kernel, C = 10):
    samples, features = X.shape
    K = np.zeros((samples, samples))
    for i in range(samples):
        for j in range(samples):
            K[i, j] = kernel(X[i], X[j])
    H = np.outer(y, y) * K

    # Converting into cvxopt format
    P = cvxopt_matrix(H)
    q = cvxopt_matrix(-1 * np.ones((samples)))
    G = cvxopt_matrix(np.vstack((np.diag(np.ones(samples)) * -1, np.identity(samples))))
    h = cvxopt_matrix(np.hstack((np.zeros(samples), np.ones(samples) * C)))
    A = cvxopt_matrix(y, (1, samples), 'd')
    b = cvxopt_matrix(0.0)

    #solve optimization problem for quadratic programming
    cvxopt_solvers.options['show_progress'] = False
    sol = cvxopt_solvers.qp(P, q, G, h, A, b)
    alphas = np.ravel(sol['x'])

    # Selecting the set of indices S corresponding to non zero parameters
    S = alphas > 1e-5

    support_vectors = X[S]
    support_alphas = alphas[S]
    support_vector_y = y[S]


    w = ((y * alphas).T @ X).reshape(-1, 1)
    b = np.mean(support_vector_y - np.dot(support_vectors, w))


    #print('Alphas = ', support_alphas)
    #print('w = ', w.flatten())
    #print('b = ', b)

    plot_data_with_support_vectors(X, y, support_vectors, C)
    show_SMV_Plot(w, b, X, y, support_vectors, kernel.__name__, C)
```

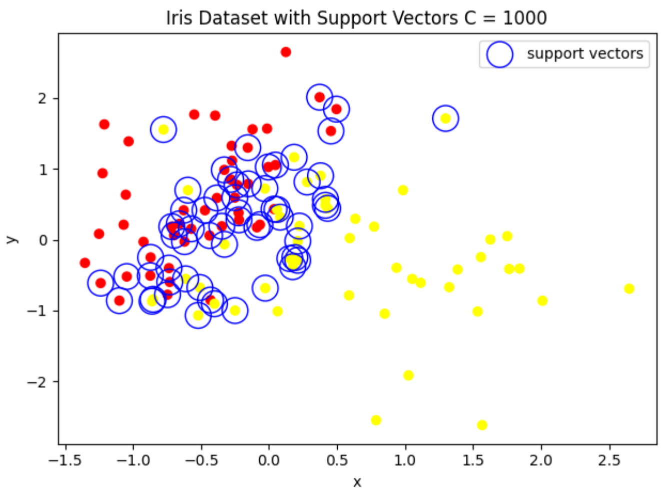
Listing 7: Optimization of SVM

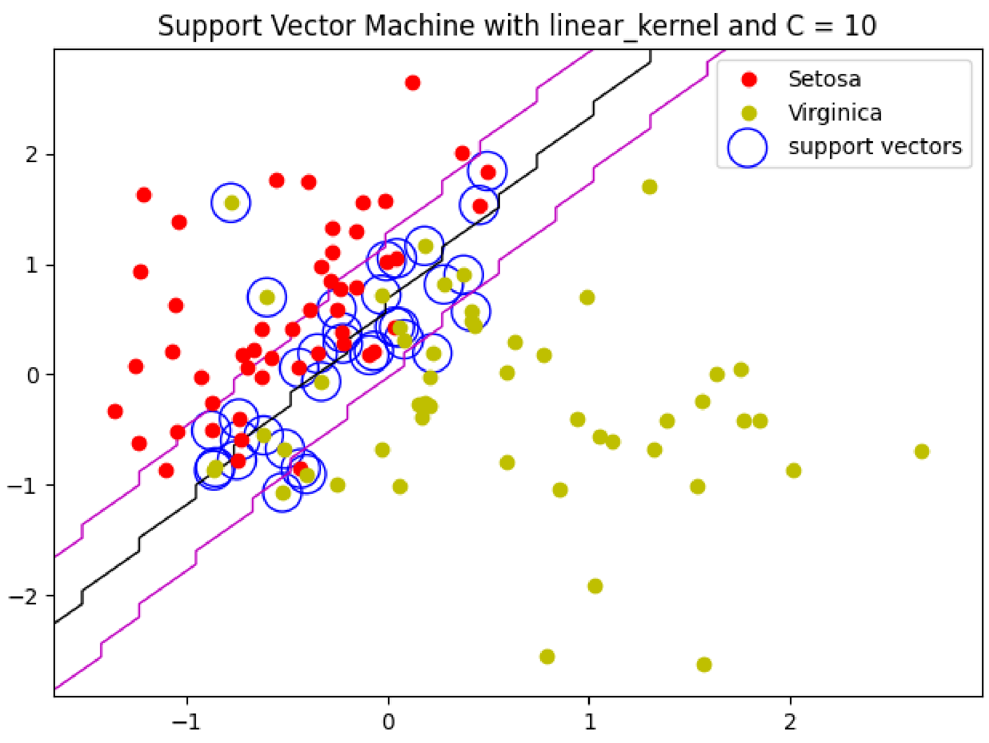
Evaluating different cost-factors C for the misclassification and get the perfect value that fits the dataset, we have to split the dataset into a Trainingset and Testset and cross-validate it on the data. Calculate the Error and interpret the results (not doing it here, we only say that C=10 fits at best):

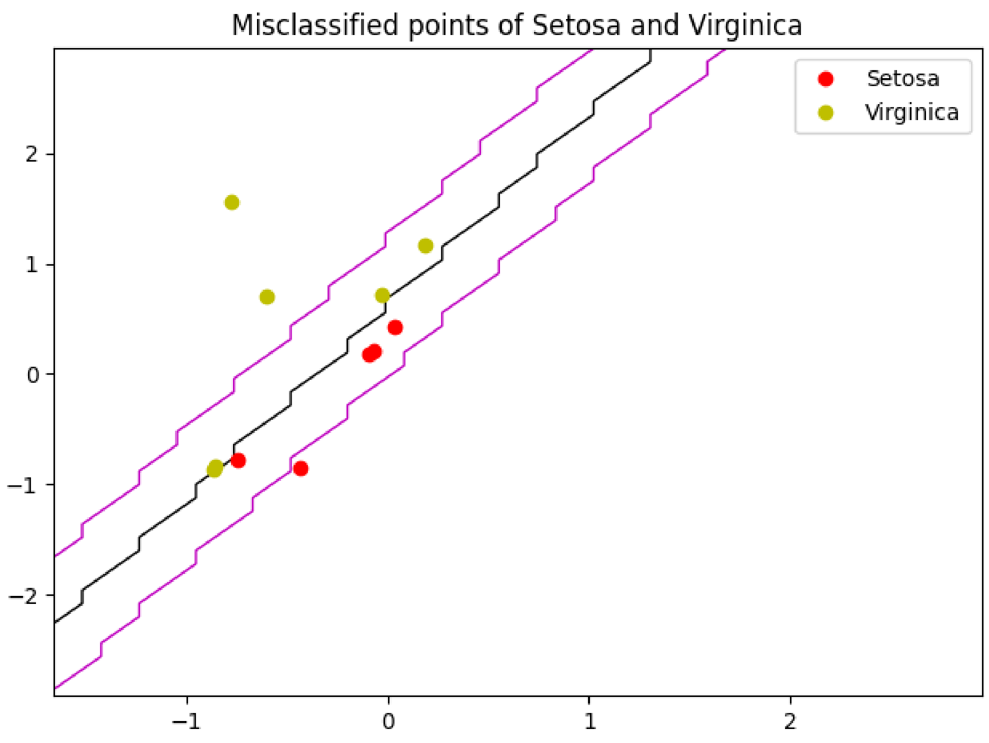












```python
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix as cvxopt_matrix
from cvxopt import solvers as cvxopt_solvers

iris = np.genfromtxt('dataSets/iris-pca.txt')
X_data, y_data = iris[:, 0:2], iris[:, 2]
```

```python

def show_data(X, y):
    plt.figure()
    plt.title('Iris Dataset --> red=Setosa, yellow=Virginica')
    plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

def linear_kernel(x, y):
    return np.dot(x, y)

def plot_data_with_support_vectors(X, y, support_vectors, C):
    plt.figure()
    plt.title('Iris Dataset with Support Vectors C = {}'.format(C))
    plt.scatter(X[:, 0], X[:, 1], cmap='autumn', c=y_data)
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1], label='support vectors', facecolors='none', linewidths=1,
                edgecolors='b', s=300)
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

def doSMV(X, y, kernel, C = 10):
    samples, features = X.shape
    K = np.zeros((samples, samples))
    for i in range(samples):
        for j in range(samples):
            K[i, j] = kernel(X[i], X[j])
    H = np.outer(y, y) * K

    # Converting into cvxopt format
    P = cvxopt_matrix(H)
    q = cvxopt_matrix(-1 * np.ones((samples)))
    G = cvxopt_matrix(np.vstack((np.diag(np.ones(samples)) * -1, np.identity(samples))))
    h = cvxopt_matrix(np.hstack((np.zeros(samples), np.ones(samples) * C)))
    A = cvxopt_matrix(y, (1, samples), 'd')
    b = cvxopt_matrix(0.0)

    #solve optimization problem for quadratic programming
    cvxopt_solvers.options['show_progress'] = False
    sol = cvxopt_solvers.qp(P, q, G, h, A, b)
    alphas = np.ravel(sol['x'])

    # Selecting the set of indices S corresponding to non zero parameters
    S = alphas > 1e-5

    support_vectors = X[S]
    support_alphas = alphas[S]
    support_vector_y = y[S]


    w = ((y * alphas).T @ X).reshape(-1, 1)
    b = np.mean(support_vector_y - np.dot(support_vectors, w))


    #print('Alphas = ', support_alphas)
    #print('w = ', w.flatten())
    #print('b = ', b)

    plot_data_with_support_vectors(X, y, support_vectors, C)
    show_SMV_Plot(w, b, X, y, support_vectors, kernel.__name__, C)

def predict(w, X, b, margin = 0):
    return np.sign(np.dot(X, w) + b + margin)

def seperate_data(X, labels):
    return X[labels == -1], X[labels == 1]

def show_SMV_Plot(w, b, X, y, support_vectors, kernel_name, C):
    y_pred = predict(w, X, b)
    correct = np.sum(np.equal(y_pred, y.reshape(-1, 1)))
    print("{} out of {} predictions are correct".format(correct, len(y_pred)))

    X1, X2 = seperate_data(X, y)
    plt.plot(X1[:, 0], X1[:, 1], "ro", label='Setosa')
    plt.plot(X2[:, 0], X2[:, 1], "yo", label='Virginica')
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1], label='support vectors', facecolors='none', linewidths=1,
     edgecolors='b', s=300)

    X1, X2 = np.meshgrid(np.linspace(min(X[:, 0]) - .3, max(X[:, 0]) + .3, 50), np.linspace(min(X[:, 1]) - .3, max(X[:,
     1]) + .3, 50))
    X = np.array([[x1, x2] for x1, x2 in zip(np.ravel(X1), np.ravel(X2))])
    Z = predict(w, X, b).reshape(X1.shape)
    plt.contour(X1, X2, Z, [0.0], colors='k', linewidths=1, origin='lower')
    plt.contour(X1, X2, predict(w, X, b, 1).reshape(X1.shape), [0.0], colors='m', linewidths=1, origin='lower', )
    plt.contour(X1, X2, predict(w, X, b, -1).reshape(X1.shape), [0.0], colors='m', linewidths=1, origin='lower')
    plt.title('Support Vector Machine with ' + kernel_name + ' and C = {}'.format(C))
    plt.legend()
    plt.show()

    #missclassified points
    setosa, virginica = [], []
    for i, j, k in zip(y, y_pred, X_data):
        if(i != j):
            if i == -1:
                setosa.append(k)
            else:
                virginica.append(k)
    setosa = np.asarray(setosa)
    virginica = np.asarray(virginica)
    print(setosa, virginica)

    plt.figure()
    plt.plot(setosa[:, 0], setosa[:, 1], "ro", label='Setosa')
    plt.plot(virginica[:, 0], virginica[:, 1], "yo", label='Virginica')
    plt.contour(X1, X2, Z, [0.0], colors='k', linewidths=1, origin='lower')
    plt.contour(X1, X2, predict(w, X, b, 1).reshape(X1.shape), [0.0], colors='m', linewidths=1, origin='lower', )
    plt.contour(X1, X2, predict(w, X, b, -1).reshape(X1.shape), [0.0], colors='m', linewidths=1, origin='lower')
    plt.legend()
    plt.title('Missclassified points of Setosa and Virginica')
    plt.show()


def main():
    def change_labels(labels):
        Y = []
        for y in labels:
            if y == 0:
                Y.append(-1)
            else:
                Y.append(1)
        return np.asarray(Y)

    X, y = X_data, change_labels(y_data)
    show_data(X, y)
    doSMV(X, y, linear_kernel, 10)

if __name__ == '__main__':
    main()
```

Listing 8: Full Code

3 Gaussian Processes

3.1 GP Regression

Target function $y = \sin(x) + \sin^2(x)$ with $x \in [0, 0.005, 0.01, 0.015, \ldots, 2\pi]$
The generative model: $t_n = y(x_n) + \epsilon_n$, $\epsilon \sim \mathcal{N}\left(0, \sigma_n^2\right)$.
The prediction for new data points:$p\left(t_{n+1}\right) = \mathcal{N}\left(t_{n+1} | \mathbf{0}, \mathbf{C}_{n+1}\right)$ where

$$\mathbf{C}_{n+1} = \begin{pmatrix} \mathbf{C}_n & \mathbf{k} \\ \mathbf{k} & c \end{pmatrix}$$
$$C_n = k\left(x_n, x_n\right) + \sigma_n^2 \delta_{ij}$$
$$\mathbf{k} = \left[k\left(x_1, x_{n+1}\right) \ldots k\left(x_n, x_{n+1}\right)\right]^\top$$
$$c = k\left(x_{n+1}, x_{n+1}\right) + \beta^{-1}$$

The squared exponential kernel

$$k\left(\mathbf{x}_i, \mathbf{x}_j\right) = \sigma_f^2 \exp\left(-\frac{1}{2l^2}\left(x_i - x_j\right)^2\right)$$

where σ_f^2 is the signal variance, l is the length-scale and σ_n^2 is the noise variance.
The prediction equations for the new data points:

$$m\left(x_{n+1}\right) = \mathbf{k}^\top \mathbf{C}_n^{-1} t$$
$$\sigma^2\left(x_{n+1}\right) = c - \mathbf{k}^\top \mathbf{C}_n^{-1} \mathbf{k}$$

Figure 4 shows the prediction results for iterations 1,2,4,8, and 16. Every iteration, the target point with the highest uncertainty is sampled.

```python
import numpy as np
import matplotlib.pyplot as plt

def kernel(xi, xj, sigma_f=1.0, l=1):
    return sigma_f * np.exp(-(np.linalg.norm(xi - xj)**2) / (2 * l**2))

def f(x):
    return np.sin(x) + np.sin(x)**2

def Cnew_comps(x, xnew, sigma_f=1.0, l=1): #Calculate the components of of Cnew
    n = x.shape[0]
    nnew = xnew.shape[0]

    Cn = np.empty((n, n))
    for i in range(n):
        for j in range(n):
            Cn[i,j] = kernel(x[i], x[j], sigma_f=sigma_f, l=l)

    k = np.empty((nnew, n))
    for i in range(nnew):
        for j in range(n):
            k[i,j] = kernel(xnew[i], x[j], sigma_f=sigma_f, l=l)

    c = np.empty((nnew, nnew))
    for i in range(nnew):
        for j in range(nnew):
            c[i,j] = kernel(xnew[i], xnew[j], sigma_f=sigma_f, l=l)

    return (Cn, k, c)

def gpr_params(Cn, c, k, sigma_n, t): # gaussian regression parameters
    n = Cn.shape[0]
    # mean
    m = np.dot(k, np.dot(np.linalg.inv(Cn + (sigma_n**2)*np.eye(n)), t.reshape([n, 1])))
    # Covariance.
    S = c - np.dot(k, np.dot(np.linalg.inv(Cn + (sigma_n**2)*np.eye(n)), k.T))
    return (m, S)

np.random.seed(1)
# prediction points
xnew = np.arange(0,2,0.005) * np.pi
```

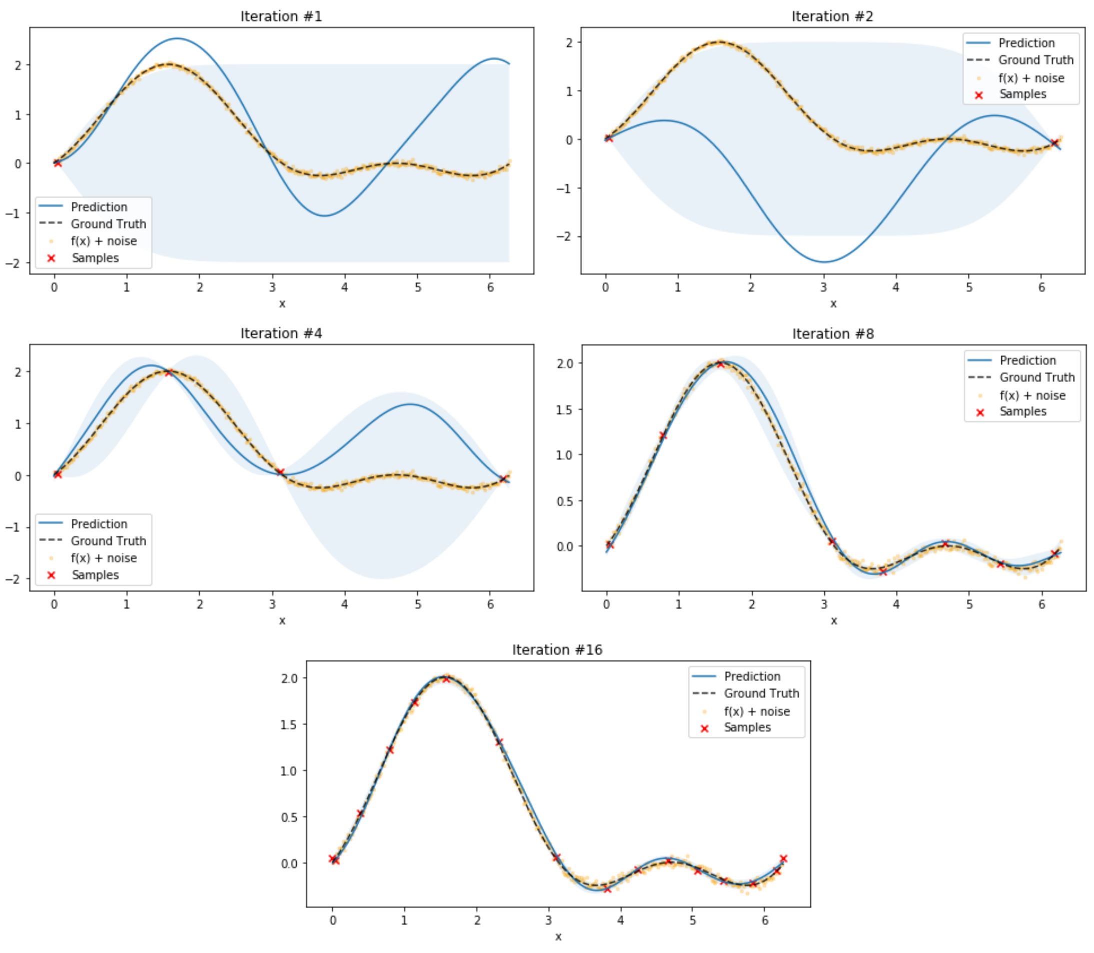


Figure 4: Gaussian process regression, using $l = 1, \sigma_f = 1$. Shaded: mean$\pm 2\times$standard deviation (uncertainty).

```python
nnew = xnew.shape[0]
# error standard deviation
sigma_n = np.sqrt(0.001)
# draw error randomly
epsilon = np.random.normal(0, sigma_n, nnew)
# observed
t_observed = f(xnew) + epsilon

# start with no target data point
x = np.array([])
t = np.array([])
# start with mean=0
mnew = np.zeros((1,1))
uncertainty = np.random.rand(nnew)
for i in range(16):
    newptidx = np.argmax(uncertainty)
    x = np.append(x,xnew[newptidx])
    t = np.append(t,t_observed[newptidx])
    n = x.shape[0]

    # Cov matrix
    l = 1
    sigma_f = 1
    Cn, k, c = Cnew_comps(x, xnew, sigma_f=sigma_f, l=l)
    Cnew_left = np.concatenate((Cn + (sigma_n**2)*np.eye(n), k), axis=0)
    Cnew_right = np.concatenate((k.T, c), axis=0)
    Cnew = np.concatenate((Cnew_left, Cnew_right), axis=1)
    # GPR parameters
    mnew, covnew = gpr_params(Cn, c, k, sigma_n, t)
    # Sample from multivariate Gaussian with the GPR parameters
    t_pred = np.random.multivariate_normal(mean=mnew.ravel(), cov=covnew)
    # Define uncertainty=2stdev
    uncertainty = 2 * np.sqrt(np.diag(covnew))

    if i in [0,1,3,7,15]:
        plt.figure(figsize=(8,4))
        plt.plot(xnew, t_pred, label='Prediction')
        plt.fill_between(xnew, mnew.ravel() + uncertainty, mnew.ravel() - uncertainty, alpha=0.1)
        plt.plot(xnew, f(xnew), '--', color='black', label="Ground Truth", alpha=0.8)
        plt.scatter(xnew, t_observed, color='orange', marker='.', linewidths=0.5, alpha=0.5, label="f(x) + noise")
        plt.scatter(x, t, color='red', marker='x', linewidths=5, label='Samples')
        plt.xlabel('x')
        plt.title("Iteration #" + str(i+1))
        plt.legend()
        plt.show()
```

Listing 9: Gaussian Process Regression.