tags: `Machine Learning`

ML HW5: Gaussian Process and SVM

I. Gaussian Process

a. code with detailed explanations

- Part1
  - **Kernel**
    - The rational quadratic kernel I used is **Radial basis function kernel with additional hyperparameter**. The equation is:
      - $kernel = \sigma * exp(-\gamma||x - x'||_2^2)$
    - The additional parameter σ is used to scale the value of RBF kernel, trying to make it more flexible.
    -
```
def kernel(X_M, X_N, sigma=1.0, gamma=1.0):
    """
    kernel with quadratic term, linear term, and constant.
    Args:
        X_M: Array of M values.
        X_N: Array of N values.
        sigma: hyper-parameter for RBF kernel.
        gamma: hyper-parameter for quadratic term.

    Return Value:
        (M x N) matrix.
    """
    MxN = np.matmul(X_M, np.transpose(X_N))
    RBF_k = sigma * np.exp((-1) * gamma * (X_M ** 2 + np.sum(X_N ** 2, axis=1) - 2 * MxN))
    return RBF_k
```
  - **Gaussian Process Regression and Prediction**
    - In Gaussian Process, we need to **construct three covariance matrices by kernel** first. Each of them defines the following relations:
      - Covar: The relation between all training data points.
      - Covar_t: The relation between all training data points and all testing data points.
      - Covar_tt: The relation between all testing data points.
    - Then we can use the above covariance matrices, y value of training datas, and variance of noise to calculate the predictive means and variances since we know that the conditional distribution p(y*|**y**) is a Gaussian distribution with:
      - $\mu(x^*) = k(x, x^*)^T * C_N^{-1} * y$
      - $\sigma^2(x^*) = k^* - k(x, x^*)^T * C_N^{-1} * k(x, x^*)$
      - where Covar = C_N, Covar_t = $k(x, x^*)$, and Covar_tt = k^* = $k(x^*, x^*) + \beta^{-1}$

```python
def prediction(Xtrain, Xtest, Ytrain, noise_var=1.0, sigma=1.0, gamma=1.0):
    """
    Prediction by Gaussian Process Regression.
    Args:
        Xtrain: Array of training data for X values. (M x 1)
        Xtest: Array of testing data for X values.   (N x 1)
        Ytrain: Array of training data for Y values. (M x 1)
        noise_var: Variance of noise.
        sigma, gamma: hyper-parameters for kernel.

    Return Value:
        mean: Array of mean values of each data in Xtest. (N x 1)
        variance: Array of variance values of each data in Xtest. (N x N)
    """
    ## Covar: The relation between all training data points (M x M)
    ## Covar_t: The relation between all training data points and all testing data points (M x N)
    ## Covar_tt: The relation between all testing data points (N x N)
    Covar = kernel(Xtrain, Xtrain, sigma, gamma) + noise_var * np.identity(Xtrain.shape[0])
    Covar_t = kernel(Xtrain, Xtest, sigma, gamma)
    Covar_tt = kernel(Xtest, Xtest, sigma, gamma) + noise_var * np.identity(Xtest.shape[0])
    Covar_inv = np.linalg.inv(Covar)

    means = np.matmul(np.matmul(np.transpose(Covar_t), Covar_inv), Ytrain)
    variances = Covar_tt -  np.matmul(np.matmul(np.transpose(Covar_t), Covar_inv), Covar_t)

    return means, variances
```

- **Train and Get Prediction**
  - Use the default hyperparameters with Training data and Testing data to train a model, and get **the predictive means and variances**.

```python
    ### Training and Prediction ###
    noise_var = 1/5
    Xtest = np.arange(start=-60, stop=60, step=0.1).reshape(-1, 1)
    if Mode == 0:
        sigma, gamma = 1.0, 1.0
        means, variances = prediction(Xtrain, Xtest, Ytrain, noise_var, sigma, gamma)
```

- **Calculate 95% confidence interval**
  - The standard score for **95% confidence level** is **1.96**, and we can multiply it by **standard deviation** to get uncertain scale. Then we use it to calculate the **confidence limit** and get the **95% confidence interval**.
  - The 95% CI for $\mu = \mu \pm 1.96 * \sigma$

```python
    CI = np.sqrt(np.diag(variances)) * 1.96
    Xtest = Xtest.flatten()
    means = means.flatten()
    UpperBound, LowerBound = means + CI, means - CI
```

- **Visualization**
  - I Use **matplotlib.pyplot** to mark **the region of the confidence interval**, **predictive mean**, and **training data**, so we can know how our model performs.

```python
    plt.plot(Xtest, means, color='blue')
    plt.fill_between(Xtest, UpperBound, LowerBound, where= UpperBound >= LowerBound, \
    facecolor = "yellow", edgecolor = "blue")
    plt.plot(Xtrain, Ytrain, "o", color='green', markersize=5)

    plt.title('hyperparam: \nsigma = {:.6f}, gamma = {:.6f}'.format(sigma, gamma))
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.xlim(-60, 60)
    plt.show()
```

- Part2
  - **Negative Marginal Log Likelihood**

- We know that the marginal likelihood is a **Gaussian distribution with mean = 0 and covariance =** C.
  - $\frac{1}{\sqrt{(2\pi)^k|\Sigma|}}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} = \frac{1}{\sqrt{(2\pi)^k|C|}}e^{-\frac{1}{2}(y-0)^TC^{-1}(y-0)}$
- In order to **optimize the hyperparameter of kernel**, we need to get the equation of **negative marginal log likelihood** and minimize it.
  - $-ln\ p(y|\theta) = \frac{1}{2}ln|C_\theta| + \frac{k}{2}ln(2\pi) + \frac{1}{2}y^TC_\theta^{-1}y$
- By computing the kernel, we can get the covariance C, which is the relation between all training data points.

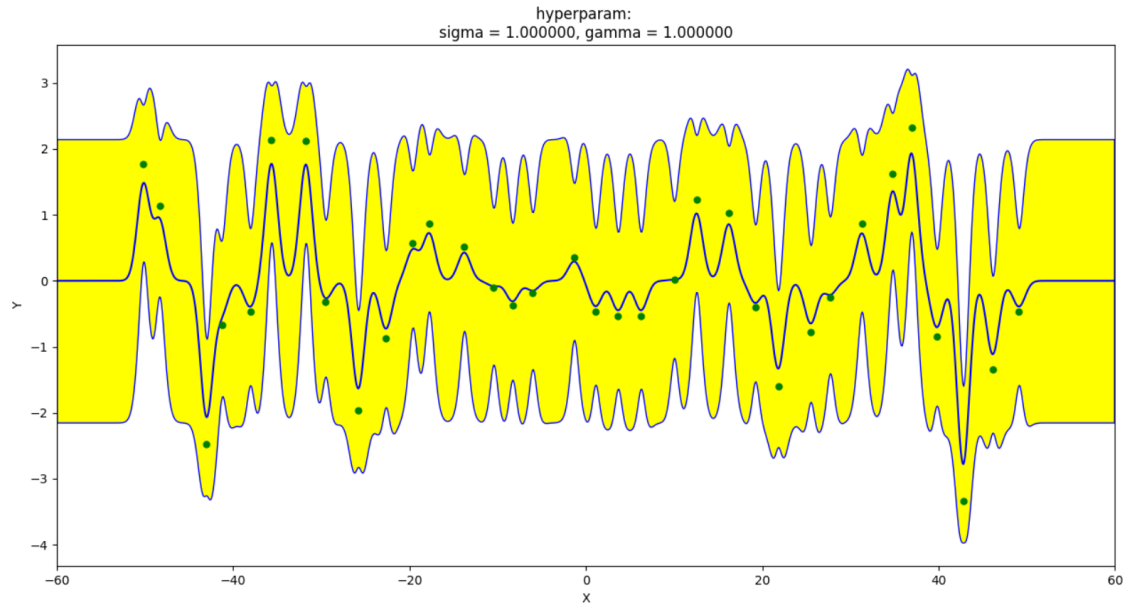
```python
def NegativeMarginalLogLikelihood(param, Xtrain, Ytrain, noise_var):
    """
    Calculate negative marginal log likelihood.
    Args:
        param: includes sigma, gamma, which are needed to optimize.
        Xtrain: Array of training data for X values. (M x 1)
        Ytrain: Array of training data for Y values. (M x 1)
        noise_var: Variance of noise.

    Return Value:
        nmll: negative marginal log likelihood value for current parameter.
    """
    sigma, gamma = param
    Covar = kernel(Xtrain, Xtrain, sigma, gamma) + noise_var * np.identity(Xtrain.shape[0])
    Covar_inv = np.linalg.inv(Covar)
    Ytrain_t = np.transpose(Ytrain)
    nmll = (1/2) * np.log(np.linalg.det(Covar)) + (Xtrain.shape[0] / 2) * np.log(2 * np.pi)
    nmll += (1/2) * (np.matmul(np.matmul(Ytrain_t, Covar_inv), Ytrain)[0][0])
    return nmll
```

- **Optimize and Train**
  - **Make the initial guess**, and use **scipy.optimize.minimize** to minimize the negative marginal log likelihood I have constructed. Also, I set the constraint for the hyperparameters. All the hyperparameters should be inside **(1e-8, 1e8)**.
  - After minimizing, we can get **the best hyperparameters for kernel**. Then use them to **do gaussian process and prediction** again.

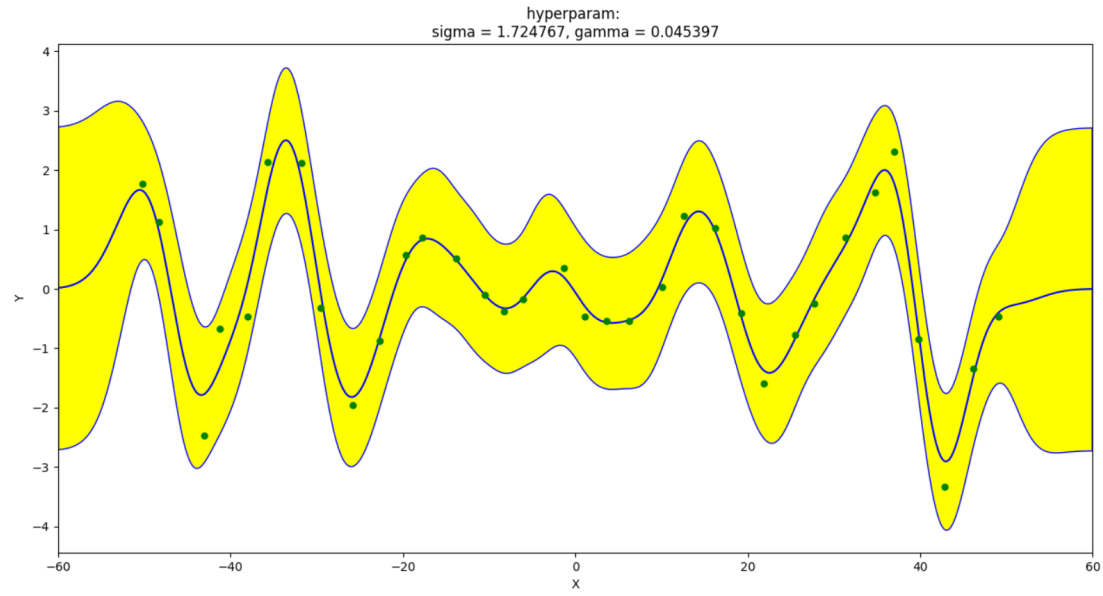
```python
    else:
        initial_guess = [1.0, 1.0]
        result = opt.minimize(fun=NegativeMarginalLogLikelihood, x0=initial_guess, \
            bounds = ((1e-8, 1e8), (1e-8, 1e8)), args = (Xtrain, Ytrain, noise_var))
        if result.success:
            sigma, gamma = result.x[0], result.x[1]
            means, variances = prediction(Xtrain, Xtest, Ytrain, noise_var, sigma, gamma)
        else:
            raise ValueError(result.message)
```

b. experiments settings and results

- Part1
  - **Set kernel hyperparameter to default value**
    - hyperparameter value is:
      - sigma = 1.000000, gamma = 1.000000



- Part2
  - **Optimize kernel hyperparameter by minimizing negative marginal log-likelihood**
    - hyperparameter value is:
      - sigma = 1.724767, gamma = 0.045397



c. observations and discussion

- For both Part1 and Part2, we can find that if the **training points have appeared in certain x value**, then we will be **more confident of guessing the y value** for that x value. That is, the **variance will be lower** when there are **training point located around**.
- However, if there is **no training data located around**, **the variance will be higher** because we have **less information** to guess the y value for that x value.
- For example, the variance around 0 is small, but the variance around 60 is large.
- For Part1, the hyperparameters for kernel function are **randomly set**, so it will **not fit the training points well**. We can find that the **curve is tortuous**, and the **variance is**

high as long as there is no training points around. Hence, this **model may be underfitting** because the kernel cannot show the correlation between training points well. However, it **gains more flexibility** to guess y value.

- For Part2, the graph is **more smooth** than Part1 because the model can **fit the training points better**. Therefore, the training points are more likely located on the mean curve, and the **variance will be smaller due to the well-defined kernel**. That is, the kernel can display the similarity relation between points well. However, **this model may be overfitting** because it **loses more flexibility** to guess y value.

II. SVM

a. code with detailed explanations

- Part1
  - **Train**
    - First, I **construct an SVM problem** by our training features and labels. Then **train an SVM model from SVM problem** by using the SVM parameter.
    -
```
def train(Xtrain, Ytrain, option='', iskernel=False):
    """
    Training a SVM model.
    Args:
        Xtrain: Features of training datas.
        Ytrain: Labels of training datas.
        iskernel: True if the kernel is user-defined.
        option: hyperparameters for training.

    Return Value:
        A SVM model if not use cross validation.
        Accuracy if use cross validation.
    """
    prob = svm_problem(Ytrain, Xtrain, iskernel)
    param = svm_parameter(option)
    res = svm_train(prob, param)
    return res
```
  - **Predict**
    - I use the SVM model which is trained by **train function**. Then **applying the testing features and labels** for the model to get accuracy.

```python
def predict(Xtest, Ytest, model):
    """
    Doing the prediction for testing data.
    Args:
        Xtest: Features of testing datas.
        Ytest: Labels of testing datas.
        model: The model which is used to predict.

    Return Value:
        p_acc: a tuple including accuracy (for classification), mean-squared
        error, and squared correlation coefficient (for regression).
    """
    return svm_predict(Ytest, Xtest, model)[1]
```

- **Different Kernel functions**
  - For each kernel function, I use **-s 0** to set the type of SVM to C-SVC (soft-margin SVM) , **-t** to define which kernel to apply (linear, polynomial, RBF kernel are corresponding to 0, 1, 2, respectively) , **-q** to enable quite mode. After setting those options, I can train three kinds of model, and make prediction by applying testing data to the models.
  - I use **Accuracy**, **Mean-Squared Error**, and **Time spent** to compare the performance of different kernel functions.

```python
if Mode == 1:
    print("-------------[Log]: Basic C-SVC for three types of kernel-------------")
    for ktype in range(0,3):
        print("Kernel Type =", kernel_type[ktype])
        print("Kernel Function =", kernel_function[ktype])
        print("SVM type = C-SVC")

        start = time.time()
        SVMmodel = train(Xtrain, Ytrain, '-s 0 -t {} -q'.format(str(ktype)), False)
        PredictRes = predict(Xtest, Ytest, SVMmodel)
        stop = time.time()

        print("Mean-Squared Error (MSE) =", PredictRes[1])
        print("Time spent = ", stop-start, "s\n", sep="")
```

- Part2
  - **cross validation train**
    - Because I set **-v** option which will enable cross validation, the **return value of train function will be the accuracy of cross validation**. Then I compare the cross validtion with the current best accuracy to **decide whether we should replace the option**.

```python
def cross_validation_train(Xtrain, Ytrain, option='', best_acc=0, best_option='', iskernel=False):
    """
    Training a SVM model with validation.
    Args:
        Xtrain: Features of training datas.
        Ytrain: Labels of training datas.
        iskernel: True if the kernel is user-defined.
        option: Hyperparameters for training.

    Return Value:
        best_acc: The current best accuracy.
        best_option: The option for best accuracy.
    """
    new_acc = train(Xtrain, Ytrain, option, iskernel)
    if (new_acc > best_acc):
        best_acc = new_acc
        best_option = option
    return best_acc, best_option
```

- **GridSearch (Not include precomputed kernel)**
  - For every kernel, set **-v 5** to implement **5-fold cross validation** when training.
  - After trying **different combination of options**, we can **figure out which combination makes the best accuracy**.

```python
def GridSearch(Xtrain, Ytrain, ktype):
    """
    Use grid search to find the best parameter for training.
    Args:
        Xtrain: Features of training datas.
        Ytrain: Labels of training datas.
        ktype: The type of the kernel.

    Return Value:
        opt_acc: The optimal accuracy in cross validation set.
        opt_option: The option for optimal accuracy.
    """
    k_fold = 5
    opt_acc = 0
    opt_option = ''
    gamma = [ 10 ** power for power in range(-3, 2) ]
    coef0 = [ 10 ** power for power in range(-1, 3) ]
    cost = [ 10 ** power for power in range(-2, 3) ]
    degree = [ deg for deg in range(2, 5) ]

    ### grid search start ###
    print("Kernel Type =", kernel_type[ktype])
    print("Kernel Function =", kernel_function[ktype])
    print("SVM type = C-SVC")
```

  - linear kernel: we only need to **iterate the cost**, which is **the importance of the slack in C-SVC**.

```python
    if ktype == 0:
        for c in cost:
            print('Current Setting: Cost = {}'.format(c))
            cur_option = '-s 0 -t {} -c {} -v {} \
                -q'.format(str(ktype), str(c), str(k_fold))
            opt_acc, opt_option = \
                cross_validation_train(Xtrain, Ytrain, cur_option, opt_acc, opt_option, False)
```

  - polynomial kernel: we need to iterate **cost**, **degree of polynomial**, **gamma** (which is used to control the scale of $u' * v$) , and coef0.

```
elif ktype == 1:
    for c in cost:
        for d in degree:
            for g in gamma:
                for coef in coef0:
                    print('Current Setting: Cost = {}, Degree = {}, Gamma = {}, Coef0 = {}'.format(c, d, g, coef))
                    cur_option = '-s 0 -t {} -c {} -d {} -g {} -r {} -v {} \
                        -q'.format(str(ktype), str(c), str(d), str(g), str(coef), str(k_fold))
                    opt_acc, opt_option = \
                        cross_validation_train(Xtrain, Ytrain, cur_option, opt_acc, opt_option, False)
```

- RBF kernel: we need to iterate **cost**, **gamma** (which is used to control the scale of $||u-v||_2^2$) .

```
elif ktype == 2:
    for c in cost:
        for g in gamma:
            print('Current Setting: Cost = {}, Gamma = {}'.format(c, g))
            cur_option = '-s 0 -t {} -c {} -g {} -v {} \
                -q'.format(str(ktype), str(c), str(g), str(k_fold))
            opt_acc, opt_option = \
                cross_validation_train(Xtrain, Ytrain, cur_option, opt_acc, opt_option, False)
```

- **Perform the best hyperparameter**
  - After getting the best hyperparameters of the kernel functions by **grid search**, I apply these options to train the models and make the prediction.

```
elif Mode == 2:
    print("-------------[Log]: Grid Search with 5-fold cross validation-------------")
    for ktype in range(0,3):
        print("--------------[Log]: Grid Search for {}--------------".format(kernel_type[ktype]))
        best_acc, best_option = GridSearch(Xtrain, Ytrain, ktype)
        print("--------------[Log]: Grid Search for {} End--------------".format(kernel_type[ktype]))
        print('Best Cross Validation Accuracy = {}%'.format(best_acc))
        print('Best option = \"{}\"'.format(best_option))
        print("Start to use best option to train...")
        best_option = best_option.replace("-v 5 ", "", 1)
        SVMmodel = train(Xtrain, Ytrain, best_option, False)
        PredictRes = predict(Xtest, Ytest, SVMmodel)
        print("Mean-Squared Error (MSE) =", PredictRes[1], "\n")
```

- Part3
  - **linear kernel**
    - I construct the linear kernel by the following equation:
      - $kernel = u * v'$

```
def linear_kernel(X_M, X_N):
    """
    kernel with linear term.
    Args:
        X_M: Array of M values.
        X_N: Array of N values.

    Return Value:
        (M x N) matrix.
    """
    return np.matmul(X_M, np.transpose(X_N))
```

  - **RBF kernel**
    - I construct the RBF kernel by the following equation:
      - $kernel = exp(-\gamma * ||u-v||_2^2)$

```python
def RBF_kernel(X_M, X_N, gamma=1.0):
    """
    RBF kernel.
    Args:
        X_M: Array of M values.
        X_N: Array of N values.
        gamma: hyper-parameter for quadratic term.

    Return Value:
        (M x N) matrix.
    """
    return np.exp((-1) * gamma * (np.sum(X_M ** 2, axis=1).reshape(-1, 1) + \
        np.sum(X_N ** 2, axis=1) - 2 * np.matmul(X_M, np.transpose(X_N))))
```

- **Grid Search (precomputed kernel)**
  - Precomputed kernel: we need to iterate **cost**, **gamma** because we **combine linear kernel with RBF kernel**. For every iteration, I compute the kernel with certain gamma, and **implement it as training features**. Also, I set **is_kernel to be true** to make sure that we use precomputed kernel when training.

```python
    else:
        linear_k = linear_kernel(Xtrain, Xtrain)
        for g in gamma:
            RBF_k = RBF_kernel(Xtrain, Xtrain, g)
            for c in cost:
                print('Current Setting: Cost = {}, Gamma = {}'.format(c, g))
                kernel = np.hstack((np.arange(1, 5001).reshape(-1,1), linear_k + RBF_k))
                cur_option = '-s 0 -t {} -c {} -g {} -v {} -q'.format(str(ktype), str(c), str(g), str(k_fold))
                opt_acc, opt_option = cross_validation_train(kernel, Ytrain, cur_option, opt_acc, opt_option, True)
                if cur_option == opt_option:
                    opt_option = '{} {} {}'.format(opt_option, c, g)

    ### grid search end ###
    return opt_acc, opt_option
```

- **Run linear kernel + RBF kernel**
  - After precomputing our kernel, I use **np.hstack** to **append the row ID in front of the kernel** because the ID for kernel matrix is needed. Then we can **use this precomputed kernel as training features** and the options obtained from **grid search** to train the model.
  - Also, I **set -t to be 4** in order to **use the user-defined kernel** when training.

```python
    elif Mode == 3:
        ktype = 4
        print("-------------[Log]: Grid Search with 5-fold cross validation-------------")
        print("-------------[Log]: Grid Search for {}-------------".format(kernel_type[ktype]))
        best_acc, best_option = GridSearch(Xtrain, Ytrain, ktype)
        print("-------------[Log]: Grid Search for {} End-------------".format(kernel_type[ktype]))
        print('Best Cross Validation Accuracy = {}%'.format(best_acc))
        print('Best option = \"{}\"'.format(best_option))
        print("Start to use best option to train...")
        gamma, cost = best_option.split()[-2], best_option.split()[-1]

        start = time.time()
        linear_k = linear_kernel(Xtrain, Xtrain)
        RBF_k = RBF_kernel(Xtrain, Xtrain, float(gamma))
        kernel = np.hstack((np.arange(1, 5001).reshape(-1,1), linear_k + RBF_k))
        SVMmodel = train(kernel, Ytrain, '-s 0 -t {} -c {} -g {} -q'.format(str(ktype), str(cost), str(gamma)), True)

        linear_test = np.transpose(linear_kernel(Xtrain, Xtest))
        RBF_test = np.transpose(RBF_kernel(Xtrain, Xtest, float(gamma)))
        kernel_test = np.hstack((np.arange(1, 2501).reshape(-1,1), linear_test + RBF_test))
        PredictRes = predict(kernel_test, Ytest, SVMmodel)
        stop = time.time()

        print("Mean-Squared Error (MSE) =", PredictRes[1])
        print("Time spent = ", stop-start, "s\n", sep="")
```

b. experiments settings and results

- Part1
  - **Use default option and C-SVC mode to train different kernel functions**
    - ```
      Kernel Type = linear
      Kernel Function = u'*v
      SVM type = C-SVC
      Accuracy = 95.08% (2377/2500) (classification)
      Mean-Squared Error (MSE) = 0.1404
      Time spent = 3.2166502475738525s

      Kernel Type = polynomial
      Kernel Function = (gamma*u'*v + coef0)^degree
      SVM type = C-SVC
      Accuracy = 34.68% (867/2500) (classification)
      Mean-Squared Error (MSE) = 2.6212
      Time spent = 27.660475969314575s

      Kernel Type = radial basis function (RBF)
      Kernel Function = exp(-gamma*|u-v|^2)
      SVM type = C-SVC
      Accuracy = 95.32% (2383/2500) (classification)
      Mean-Squared Error (MSE) = 0.1492
      Time spent = 6.672668695449829s
      ```
	linear	**polynomial**	**RBF**
Accuracy	95.08%	34.68%	95.32%
MSE	0.1404	2.6212	0.1492
Time Spent	3.2167	27.660	6.6727

- Part2
  - **Use grid search for finding parameters of the best performing model**
    - **Cost** is selected from $[10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}]$
    - **Gamma** is selected from $[10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}]$
    - **Degree** is selected from $[2, 3, 4]$
    - **Coef0** is selected from $[10^{-1}, 10^{0}, 10^{1}, 10^{2}]$
    - **Linear Kernel**
      - cost = 0.01
      - ```
        ------------[Log]: Grid Search for linear End------------
        Best Cross Validation Accuracy = 97.08%
        Best option = "-s 0 -t 0 -c 0.01 -v 5 -q"
        Start to use best option to train...
        Accuracy = 95.96% (2399/2500) (classification)
        Mean-Squared Error (MSE) = 0.1256
        ```
    - **Polynomial Kernel**
      - cost = 0.01, degree = 3, gamma = 10, coef0 = 100

```
-------------[Log]: Grid Search for polynomial End-------------
Best Cross Validation Accuracy = 98.22%
Best option = "-s 0 -t 1 -c 0.01 -d 3 -g 10 -r 100 -v 5 -q"
Start to use best option to train...
Accuracy = 97.92% (2448/2500) (classification)
Mean-Squared Error (MSE) = 0.0568
```

- **RBF Kernel**
  - cost = 100, gamma = 0.01

```
-------------[Log]: Grid Search for radial basis function (RBF) End-------------
Best Cross Validation Accuracy = 98.26%
Best option = "-s 0 -t 2 -c 100 -g 0.01 -v 5 -q"
Start to use best option to train...
Accuracy = 98.16% (2454/2500) (classification)
Mean-Squared Error (MSE) = 0.0504
```

	linear	**polynomial**	**RBF**	**linear + RBF**
Cross Validation Accuracy	97.08%	98.22%	98.26%	97.08%
Test Accuracy	95.96%	97.92%	98.16%	95.64%
MSE	0.1256	0.0568	0.0504	0.118
cost (paramter)	0.01	0.01	100	0.1
gamma (paramter)	-	10	0.01	10
degree (paramter)	-	3	-	-
coef0 (paramter)	-	100	-	-

- Part3
  - **Use linear + RBF kernel**
    - cost = 0.1, gamma = 10

```
-------------[Log]: Grid Search for precomputed kernel End-------------
Best Cross Validation Accuracy = 97.08%
Best option = "-s 0 -t 4 -c 0.1 -g 10 -v 5 -q 0.1 10"
Start to use best option to train...
Accuracy = 95.64% (2391/2500) (classification)
Mean-Squared Error (MSE) = 0.118
Time spent = 23.679781913757324s
```

    - The table for comparison is together with Part2.

c. observations and discussion

- Part1
  - For default options, **polynomial kernel cannot perform well** as same as linear kernel and RBF kernel. I think the reason is that **polynomail kernel is more**

complicated（It has more parameters）,if we don't set the parameters properly, it cannot display the similarity relation well for the training data.
    - Both linear kernel and RBF kernel has high accuracy even though we use default options.
    - **Linear kernel is faster** than other kernels due to **its simple architecture**.
- Part2
    - Compare to Part1, we can find that **polynomial kernel gains a lot of accuracy** if we find a proper parameters. The accuracy of Linear kernel and RBF kernel are also improved. Hence, I believe that **the parameters are significant for the SVM model**. If we can find the proper parameter, these three kernel can all perfrom well.
    - Because I implement **5-fold cross validation on training**, our svm models are **not easy to overfit the training datas**. Hence, the cross validation accuracy is highly related to test accuracy in this experiment.
    - The accuracy reduces a little for testing data because the optimal parameters are for training datas, not for testing datas.
    - **The slack influences a lot for RBF kernel** because the cost is high. However, it is not so important for other two kernels.
- Part3
    - Although linear+RBF kernel has high accuracy, it **doesn't perform better than linear kernel or RBF kernel** even though I apply grid search for it in this experiment. Moreover, it **takes more time** to train than other two kernels.
    - Linear+RBF kernel is more complicated, so it may be more likely to overfit. This may lead to the result that it doesn't fit better than linear kernel and RBF kernel.