3. **Kernel vs Neural Network:**

(a) (10 points) Implement and test kernel logistic regression model — complete the forward() function of the class Kernel Layer() and the init () function of the class Kernel LR(), test your implementation in "main.py".

For the parameters:

- learning_rate = 0.01
- max_epoch = 10
- batch_size = 100
- sigma = 5

Test accuracy = 0.9784

Time taken = 16.6 sec

```
class Kernel_Layer(nn.Module):

   def __init__(self, sigma, hidden_dim=None):
       """
       Set hyper-parameters.
       Args:
           sigma: the sigma for Gaussian kernel (radial basis function)
           hidden_dim: the number of "kernel units", default is None,
then the number of "kernel units"
                                      will be set to be the number of
training samples
       """
       super(Kernel_Layer, self).__init__()
       self.sigma = sigma
       self.hidden_dim = hidden_dim


   def forward(self, x):
       """
       Compute Gaussian kernel (radial basis function) of the input
sample batch
       and self.prototypes (stored training samples or "representatives"
of training samples).

       Args:
           x: A torch tensor of shape [batch_size, n_features]

       Returns:
           A torch tensor of shape [batch_size, num_of_prototypes]
```

```
        """
        assert x.shape[1] == self.prototypes.shape[1]

        #my_function = lambda row: self.kernel_function(row)

        #print(x.shape)
        #print('above is the x')
        #result = torch.Tensor(list(map(my_function, x))).float()
        #print('------')
        #print(result.shape)
        _size = (x.shape[0], self.prototypes.shape[0], x.shape[1])
        # batch size , 1, nfeatures - batch size, m, features.   1, m,
256; n, m , 256
        x = x.unsqueeze(1).expand(_size)
        p = self.prototypes.unsqueeze(0).expand(_size)
        norms = (x - p).pow(2).sum(-1).pow(0.5)
        return torch.exp(-1 * norms / (2 * (self.sigma * self.sigma)))
```

```
class Kernel_LR(nn.Module):

   def __init__(self, sigma, hidden_dim):
        """
        Define network structure.

        Args:
            sigma: used in the kernel layer.
            hidden_dim: the number of prototypes in the kernel layer,
                                                  in this model, hidden_dim has to
be equal to the
                                                  number of training samples.
        """
        super(Kernel_LR, self).__init__()
        self.hidden_dim = hidden_dim
        ### YOUR CODE HERE
        _kernel_layer = Kernel_Layer(sigma=sigma)
        linear_layer  =
nn.Linear(in_features=self.hidden_dim,out_features=1,bias=False)
        self.net = nn.Sequential(_kernel_layer,linear_layer)
        ### END YOUR CODE
```

(b) (15 points) Implement and test radial basis function network model — complete the k means() function of the class Kernel Layer() and the init () function of the class RBF(), test your implementation in "main.py".

For the parameters:

- hidden_dim = 12
- learning_rate = 0.01
- max_epoch = 10
- batch_size = 100
- sigma = 5

Test accuracy = 0.9632
Time taken = 0.98 sec

```
class Kernel_Layer(nn.Module):

   def __init__(self, sigma, hidden_dim=None):
       """
       Set hyper-parameters.
       Args:
           sigma: the sigma for Gaussian kernel (radial basis function)
           hidden_dim: the number of "kernel units", default is None,
then the number of "kernel units"
                                      will be set to be the number of
training samples
       """
       super(Kernel_Layer, self).__init__()
       self.sigma = sigma
       self.hidden_dim = hidden_dim


   def _k_means(self, X):
       """
       K-means clustering

       Args:
           X: A Numpy array of shape [n_samples, n_features].

       Returns:
           centroids: A Numpy array of shape [self.hidden_dim,
n_features].
       """
       ### YOUR CODE HERE
```

```
        kmeans_model = KMeans(init='random', n_clusters=self.hidden_dim)
        kmeans_model.fit(X)

        centroids = kmeans_model.cluster_centers_
        ### END YOUR CODE
        return centroids
```

```
class RBF(nn.Module):

   def __init__(self, sigma, hidden_dim):
        """
        Define network structure.

        Args:
            sigma: used in the kernel layer.
            hidden_dim: the number of prototypes in the kernel layer,
                                       in this model, hidden_dim is a
user-specified hyper-parameter.
        """
        super(RBF, self).__init__()
        self.sigma = sigma
        self.hidden_dim = hidden_dim
        _kernel_layer =
Kernel_Layer(sigma=self.sigma,hidden_dim=self.hidden_dim)
        linear_layer = nn.Linear(in_features=self.hidden_dim,
out_features=1, bias=False)
        self.net = nn.Sequential(_kernel_layer, linear_layer)
```

(c) (15 points) Implement and test feed forward neural network model — complete the init () function of the class FFN(), test your implementation in “main.py”.

For the parameters:

- hidden_dim = 12
- learning_rate = 0.01
- max_epoch = 10
- batch_size = 100

Test accuracy = 0.989
Time taken = 0.156 sec

```
class FFN(nn.Module):

   def __init__(self, input_dim, hidden_dim):
       """
       Define network structure.

       Args:
           input_dim: number of features of each input.
           hidden_dim: the number of hidden units in the hidden layer, a
user-specified hyper-parameter.
       """
       super(FFN, self).__init__()
       ### YOUR CODE HERE
       # Use pytorch nn.Sequential object to build a network composed of
       # two linear layers (nn.Linear object)
       l1 = nn.Linear(in_features=input_dim, out_features=hidden_dim,
bias=False)
       l2 = nn.Linear(in_features=hidden_dim, out_features=1,
bias=False)
       self.net = nn.Sequential(l1,l2)

       ### END CODE HERE
```

We see that the Feed forward network takes the least time. All the three models result in good accuracy with the best being given by feed forward network. One reason that feed forward network works so well is that it is able to achieve at the minima quickly and it also has computationally lower operations as compared to the other both.

4. PCA vs Autoencoder:

(a) (10 points) In the class PCA(), complete the do pca() function.

```
def _do_pca(self):
   '''
   To do PCA decomposition.
   Returns:
       Up: Principal components (transform matrix) of shape [n_features,
n_components].
       Xp: The reduced data matrix after PCA of shape [n_components,
n_samples].
   '''
   ### YOUR CODE HERE
```

```
    batch_size = self.X.shape[1]

    mean = (1 / batch_size) * self.X @ np.ones((batch_size, 1))
    X_centered = self.X - mean @ np.ones((batch_size, 1)).T
    u, _, _ = np.linalg.svd(X_centered)
    Up = u[:,:self.n_components]
    Xp = Up.T @ self.X
    ### END YOUR CODE
    return Up, Xp
```

(b) (5 points) In the class PCA(), complete the reconstruction() function to perform data reconstruction. Please evaluate your code by testing different numbers of the principal component that p = 32, 64, 128.

PCA-Reconstruction error for 32 components is 134.91234205467669
PCA-Reconstruction error for 64 components is 87.07439083700217
PCA-Reconstruction error for 128 components is 46.16377547729022

```
def reconstruction(self, Xp):
    '''
    To reconstruct reduced data given principal components Up.

    Args:
    Xp: The reduced data matrix after PCA of shape [n_components,
n_samples].

    Return:
    X_re: The reconstructed matrix of shape [n_features, n_samples].
    '''
    ### YOUR CODE HERE
    X_re = self.Up @ Xp

    ### END YOUR CODE
    return X_re
```

(c) (10 points) In the class AE(), complete the network() and forward() function. Please follow the note (http://people.tamu.edu/~sji/classes/PCA.pdf) to implement your network. Note that for problems (c), (e), and (f), the weights need to be 3 shared between the encoder and the decoder with weight matrices transposed to each other

```python
def _network(self):
    '''

    You are free to use the listed functions and APIs from torch or
torch.nn:
        torch.empty
        nn.Parameter
        nn.init.kaiming_normal_

    You need to define and initialize weights here.

    '''

    ### YOUR CODE HERE

    '''
    Note: you should include all the three variants of the networks here.
    You can comment the other two when you running one, but please
include
    and uncomment all the three in you final submissions.
    '''

    # Note: here for the network with weights sharing. Basically you need
to follow the
    #p = feautes,k=hidden dimension
    wts = torch.empty(self.n_features,self.d_hidden_rep)
    wts = nn.init.kaiming_normal_(wts)
    self.shared_wts = nn.Parameter(wts)
    self.w = self.shared_wts

    # Note: here for the network without weights sharing

    # self.l1 = nn.Linear(self.n_features,self.d_hidden_rep,bias=False)
    # self.l2 = nn.Linear(self.d_hidden_rep,self.n_features,bias=False)
    # self.second_mode_seq = nn.Sequential(self.l1,self.l2)
    # self.w = self.l1.weight
    # # Note: here for the network with more layers and nonlinear
functions
    # self.in1 =
nn.Linear(self.n_features,2*self.d_hidden_rep,bias=False)
    # self.in2 =
```

```
nn.Linear(self.d_hidden_rep*2,self.d_hidden_rep,bias=False)
   # self.out1 =
nn.Linear(self.d_hidden_rep,2*self.d_hidden_rep,bias=False)
   # self.out2 =
nn.Linear(self.d_hidden_rep*2,self.n_features,bias=False)
   # self.third_mode_seq = nn.Sequential(self.in1,nn.ReLU(),
   #                                     self.in2,nn.ReLU(),
   #                                     self.out1,nn.ReLU(),
   #                                     self.out2,nn.Sigmoid()
   #                                     )
   #
   #
   ### END YOUR CODE

def _forward(self, X):
   '''

   You are free to use the listed functions and APIs from torch and
torch.nn:
       torch.mm
       torch.transpose
       nn.Tanh
       nn.ReLU
       nn.Sigmoid

   Args:
       X: A torch tensor of shape [n_features, batch_size].
           for input images.

   Returns:
       out: A torch tensor of shape [n_features, batch_size].

   '''

   ### YOUR CODE HERE

   '''
   Note: you should include all the three variants of the networks here.
   You can comment the other two when you running one, but please
include
   and uncomment all the three in you final submissions.
```

```
    '''

    # Note: here for the network with weights sharing. Basically you need 
to follow the
    # formula (WW^TX) in the note at 
http://people.tamu.edu/~sji/classes/PCA.pdf .

    first_layer = torch.mm(torch.transpose(self.shared_wts,0,1),X)
    last_layer = torch.mm(self.shared_wts,first_layer)

    # Note: here for the network without weights sharing
    #last_layer = self.second_mode_seq(X.t()).t()

    # Note: here for the network with more layers and nonlinear functions
    #last_layer = self.third_mode_seq(X.t()).t()

    ### END YOUR CODE
    return last_layer
```

(d) (5 points) In the class AE(), complete the reconstruction() function to perform data reconstruction. Please test your function using three different dimensions for the hidden representation d that d = 32, 64, 128

AE-Reconstruction error for 32-dimensional hidden representation is 131.01395589037753
AE-Reconstruction error for 64-dimensional hidden representation is 86.58451799931687
AE-Reconstruction error for 128-dimensional hidden representation is 46.736346232419876

```
def reconstruction(self, X):
    '''
    To reconstruct data. You're required to reconstruct one by one here,
    that is to say, for one loop, input to the network is of the shape 
[n_features, 1].
    Args:
        X: The data matrix with shape [n_features, n_any], a numpy array.
    Returns:
        X_re: The reconstructed data matrix, which has the same shape as 
X, a numpy array.
    '''
    _, n_samples = X.shape
    output = []
```

```
    with torch.no_grad():
        for i in range(n_samples):
            ### YOUR CODE HERE

            # Note: Format input curr_X to the shape [n_features, 1]

            curr_X = np.expand_dims(X[:, i], axis=1)
            ### END YOUR CODE
            curr_X_tensor = torch.tensor(curr_X).float()
            curr_X_re_tensor = self._forward(curr_X_tensor)
            output.append(curr_X_re_tensor.numpy())
            ### YOUR CODE HERE

            # Note: To achieve final reconstructed data matrix with the
shape [n_features, n_any].
    X_re =np.asarray(output).T
        ### END YOUR CODE
    return X_re
```

(e) (10 points) Compare the reconstruction errors from PCA and AE. Note that you need to set p = d for comparisons. Please evaluate the errors using p = d = 32, 64, 128. Report the reconstruction errors and provide a brief analysis

If we observe the AE and PCA reconstruction errors, we find that for each of the dimensions they are similar. But AE takes longer time, as it involves more parameters and higher computations. This also indicates that neural network in AE performs dimensionality reduction similar to PCA

(f) (10 points) Experimentally justify the relations between the projection matrix G in PCA and the optimized weight matrix W in AE. Note that you need to set p = d for valid comparisons. Please explore three different cases that p = d = 32, 64, 128. We recommend to first use frobeniu norm error() to verify if W and G are the same. If not, please follow the note (http://people.tamu.edu/~sji/classes/PCA. pdf) to implement necessary transformations for two matrices G and W and explore the relations. You need to modify the code in "main.py".

Comparing projection matrix of PCA, G and optimised matrix of AE, **W** through frobeniu norm error()

dimension	frobeniu norm error() b/w G and W	frobeniu norm error() b/w G^TG and W^TW

32	8.058,	0.537
64	11.298	0.0168
128	16.0086	0.0239

Ideally the matrices G and W(which would be the optimal solution of W in AE) should have been same and but this is not the case due to small differences in the result. The solutions to PCA are not unique and can differ by an orthogonal matrix. This is the main reason why the G and W look so different. But when we computed $G^T G$ and $W^T W$, the effect of the orthogonal matrix disappears, and hence we see very less value of the frobeniu norm

```
if __name__ == '__main__':
   dataloc = "../data/USPS.mat"
   A = load_data(dataloc)
   A = A.T
   ## Normalize A
   A = A/A.max()

   ### YOUR CODE HERE
   # Note: You are free to modify your code here for debugging and
justifying your ideas for 5(f)
   ps = [32, 64, 128]#[64]#,64,128] #, 64, 128]#[50, 100, 150]
   e1 = []
   e2 = []

   for p in ps:
       G = test_pca(A, p)
       final_w = test_ae(A, p)

       e1.append(frobeniu_norm_error(G,final_w))
       e2.append(frobeniu_norm_error(G.T @G,final_w.T@final_w))
   ### END YOUR CODE

   print(f'direct comparision : {e1}')
   print(f'Indirect comparision : {e2}')
```

(g) (10 points) Please modify the network() and forward() function so that the weights are not shared between the encoder and the decoder. Report the reconstructions errors for d = 32, 64, 128. Please compare with the sharing weights case and briefly analyze you results.

In the current case,

AE-Reconstruction error for 32-dimensional hidden representation is 129.68845193125063
AE-Reconstruction error for 64-dimensional hidden representation is 86.09042815340115
AE-Reconstruction error for 128-dimensional hidden representation is 46.011382887349285

For sharing weights case:
AE-Reconstruction error for 32-dimensional hidden representation is 131.01395589037753
AE-Reconstruction error for 64-dimensional hidden representation is 86.58451799931687
AE-Reconstruction error for 128-dimensional hidden representation is 46.736346232419876

We can see that in both the cases the reconstruction error is similar. So we can say that even though the weights are not shared the neural network is able to replicate it .

```
def _network(self):
    '''

    You are free to use the listed functions and APIs from torch or
torch.nn:
        torch.empty
        nn.Parameter
        nn.init.kaiming_normal_

    You need to define and initialize weights here.

    '''

    ### YOUR CODE HERE

    '''
    Note: you should include all the three variants of the networks here.
    You can comment the other two when you running one, but please
include
    and uncomment all the three in you final submissions.
    '''

    # Note: here for the network with weights sharing. Basically you need
to follow the
    #p = feautes,k=hidden dimension
    # wts = torch.empty(self.n_features,self.d_hidden_rep)
    # wts = nn.init.kaiming_normal_(wts)
```

```
    # self.shared_wts = nn.Parameter(wts)
    # self.w = self.shared_wts

    # Note: here for the network without weights sharing

    self.l1 = nn.Linear(self.n_features,self.d_hidden_rep,bias=False)
    self.l2 = nn.Linear(self.d_hidden_rep,self.n_features,bias=False)
    self.second_mode_seq = nn.Sequential(self.l1,self.l2)
    self.w = self.l1.weight
    # # Note: here for the network with more layers and nonlinear
functions
    self.in1 = nn.Linear(self.n_features,2*self.d_hidden_rep,bias=False)
    self.in2 =
nn.Linear(self.d_hidden_rep*2,self.d_hidden_rep,bias=False)
    self.out1 =
nn.Linear(self.d_hidden_rep,2*self.d_hidden_rep,bias=False)
    self.out2 = nn.Linear(self.d_hidden_rep*2,self.n_features,bias=False)
    self.third_mode_seq = nn.Sequential(self.in1,nn.ReLU(),
                                        self.in2,nn.ReLU(),
                                        self.out1,nn.ReLU(),
                                        self.out2,nn.Sigmoid()
                                        )
    self.w = self.in2.weight
```

```
def _forward(self, X):
    '''

    You are free to use the listed functions and APIs from torch and
torch.nn:
        torch.mm
        torch.transpose
        nn.Tanh
        nn.ReLU
        nn.Sigmoid

    Args:
        X: A torch tensor of shape [n_features, batch_size].
            for input images.
```

```
    Returns:
        out: A torch tensor of shape [n_features, batch_size].

    '''

    ### YOUR CODE HERE

    '''
    Note: you should include all the three variants of the networks here.
    You can comment the other two when you running one, but please
include
    and uncomment all the three in you final submissions.
    '''

    # Note: here for the network with weights sharing. Basically you need
to follow the
    # formula (WW^TX) in the note at
http://people.tamu.edu/~sji/classes/PCA.pdf .

    #first_layer = torch.mm(torch.transpose(self.shared_wts,0,1),X)
    #last_layer = torch.mm(self.shared_wts,first_layer)

    # Note: here for the network without weights sharing
    last_layer = self.second_mode_seq(X.t()).t()

    # Note: here for the network with more layers and nonlinear functions
    last_layer = self.third_mode_seq(X.t()).t()

    ### END YOUR CODE
    return last_layer
```

(h) (10 points) Please modify the network() and forward() function to include more network layers and nonlinear functions. Please set d = 64 and explore different hyperparameters. Report the hyperparameters of the best model and its reconstruction error. Please analyze and report your conclusions.

Modification code as shown above, for multiple layers and non linear functions case for different hyperparameters and hidden dimension d = 64:

Max epochs = 300; batch size = 32; AE-Reconstruction error = 53.57546312183057
Max epochs = 500; batch size = 32; AE-Reconstruction error = 49.56116541376876

Max epochs = 700; batch size = 32; AE-Reconstruction error = 47.377009678010246
Max epochs = 1000; batch size = 32; AE-Reconstruction error = 48.18031474764952

Max epochs = 300; batch size = 64; AE-Reconstruction error = 60.286887271547684
Max epochs = 500; batch size = 64; AE-Reconstruction error = 53.54858751687961
Max epochs = 700; batch size = 64; AE-Reconstruction error = 49.47445377260037
Max epochs = 1000; batch size = 64; AE-Reconstruction error = 48.98828160971371

Max epochs = 300; batch size = 128; AE-Reconstruction error = 68.54195295194681
Max epochs = 500; batch size = 128; AE-Reconstruction error = 58.45061170888476
Max epochs = 700; batch size = 128; AE-Reconstruction error = 55.16820080455307
Max epochs = 1000; batch size = 128; AE-Reconstruction error = 50.59792993701617

Best hyperparameters for d = 64 are:
Max epochs = 700; batch size = 32; AE-Reconstruction error = 47.377009678010246

And the general trend observed (except for batch size = 32)is as the number of epochs increased the reconstruction error decreased

Reconstruction error for p=64:
PCA: 87.07
AE with weights sharing: 86.09
AE without weights sharing: 86.58

More network layers and nonlinear functions: **47.377009678010246**

We see that using more network layers and non linear functions gieves lower error that both PCA and autoencoders with a single layer. The network is able to learn more complex relations when more layers and non linear functions are used and hence the data reconstruction is more effective.

HW3 : DeepLearning, CSCE - 636

SIRI PRANITHA NAMBURI
331007601

① Given, the eigen value decomposition of A

$A = Q \Lambda Q^T$ where $Q = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$

$\Lambda = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{bmatrix}$

SVD of A would be in form $A = U \tilde{\Sigma} V^T$;
U, V are orthogonal matrices, $\tilde{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}$ $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq 0$

Given A is a symmetric matrix, Q is orthogonal ~~and has~~

$A = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{12} & u_{22} & u_{32} \\ u_{13} & u_{23} & u_{33} \end{bmatrix}$

transforming the negative sign, by changing sign of relevant column in U.

$A = \begin{bmatrix} u_{11} & u_{12} & -u_{13} \\ u_{21} & u_{22} & -u_{23} \\ u_{31} & u_{32} & -u_{33} \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{12} & u_{22} & u_{32} \\ u_{13} & u_{23} & u_{33} \end{bmatrix}$

transforming order of values to make sure diagonal elements of the middle matrix are in descending order.

$A = \begin{bmatrix} u_{11} & -u_{13} & u_{12} \\ u_{21} & -u_{23} & u_{22} \\ u_{31} & -u_{33} & u_{32} \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{13} & u_{23} & u_{33} \\ u_{12} & u_{22} & u_{32} \end{bmatrix}$ — ①

Hence equation (1) is the SVD of A with.

$$U = \begin{bmatrix} u_{11} & -u_{13} & u_{12} \\ u_{21} & -u_{23} & u_{22} \\ u_{31} & -u_{33} & u_{32} \end{bmatrix} \quad \tilde{\Sigma} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad V = \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{13} & u_{23} & u_{33} \\ u_{12} & u_{22} & u_{32} \end{bmatrix}$$

(2) PROOF OF KYFAN THERODM

$H \in R^{n\times n}$; H is symmetric matrix with eigen values $\lambda_1 \geq \dots \geq \lambda_n$; corresponding eigen vectors $U = [u_1, \dots u_n]$

we have to prove that $\lambda_1 + \dots + \lambda_k = \max_{A \in R^{n\times k};\, A^TA = I_k} \operatorname{trace}(A^THA)$

and the optimal $A^* = [u_1, \dots, u_k]Q$; Q = arbitary ortrogonal matrix

proof: ~~Here~~ let eigen decomposition of H be

$H = U\Lambda U^T$; Λ = diagnol matrix with eigen values.

$$A^T H A = A^T(U\Lambda U^T)A = A^TU\Lambda U^TA.$$

assume $A^TU = B^T$. Then $\boxed{A^THA = B^T\Lambda B}$

$$\operatorname{trace}(A^THA) = \operatorname{trace}(B^T\Lambda B)$$
$$= \sum_{n,k,j} t_{nk} \Lambda_{kj} t_{jn}$$
$$= \sum_k \left(\Lambda_{kk} \sum_n t^T_{kn} t_{nk}\right)$$

Here $t^T_{kn} t_{nk} \leq 1$ as A is semiortrogonal

and U is ortrogonal

Here A is semiorthogonal as $A^TA = I_k$; i.e;
$A^T = A^{-1}$ for $k \times k$ rows and columns, with rest being 0.

$t_{kn}^T t_{nk} = 1$, is when $v_i a_i$ is not orthogonal for
$i = 0, 1, \ldots, k$ $v_i^T a_i = 1$

and when $v_j a_j$ is orthogonal for $j = k+1, \ldots n$
$v_j^T a_j = 0$.

$$\max \operatorname{trace}(B^T \Lambda B) = \max \Big(\sum_{i=1}^{k} \lambda_i \sum_{i=1}^{k} b_i^T b_i + \sum_{j=k+1}^{n} \lambda_j \sum_{j=k+1}^{n} b_j^T b_j \Big)$$

$$= \sum_{i=1}^{k} \lambda_i = \lambda_1 + \cdots \lambda_k .$$

Hence $\lambda_1 + \cdots + \lambda_k = \max\limits_{A \in \mathbb{R}^{n \times k},\ A^TA = I_k} \operatorname{trace}(A^T H A)$

$\forall i \in [1, k]$, $\|A^T u_i\|_2 = 1 \Rightarrow u_i = A q_i$ where
$q_1, \ldots q_k$ are orthonormal

For $k+1 \le i \le n \Rightarrow \|A^T u_i\| = 0$
Hence $A^* = [u_1, \ldots u_k] Q$ where
$Q = [q_1, \ldots q_k] \in \mathbb{R}^{k \times k}$

5) Bonus Question.

let us take the training and testing kernels as K_1 and K_2 $K_1 \in \mathbb{R}^{n \times n}$, $K_2 \in \mathbb{R}^{n \times m}$.

training data: $A \in \mathbb{R}^{n \times k}$; testing data: $B \in \mathbb{R}^{m \times k}$ for some constant number of features k.

$$K_1 = AA^T \text{ ; let SVD of } K_1 = U \Sigma U^T$$

$$K_1 = U \Sigma^{1/2} (U \Sigma^{1/2})^T$$ ($\because \Sigma$ has positive values on the diagonal)

$$\therefore \boxed{A = U \Sigma^{1/2}}$$

Hence A, aka, the training data can be computed using U and Σ and it can be used for training primal solvers.

For testing, $K_2 = AB^T \Rightarrow K_2 = U \Sigma^{1/2} B^T$

$$U^T K_2 = U^T U \Sigma^{1/2} B^T$$ ($U^T U = I$ as U is orthogonal)

$$U^T K_2 = \Sigma^{1/2} B^T$$

$$B^T = (\Sigma^{1/2})^{-1} U^T K_2$$

$$\boxed{B = (\Sigma^{-1/2} U^T K_2)^T}$$

In this way, the testing data B can also be computed. We use this to get predictions.