Homework 3: SVMs & Kernel Methods

Due: Wednesday, March 1, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

In this problem set we will get up to speed with SVMs and Kernels. Long at first glance, the problem set includes a lot of helpers. You will find a review of kernalization. One section will include a revision of ridge regression which you should start to be familiar with. For the second and third problem some codes are provided to save you some time. Finally, some reminders on positive (semi)definite matrices are included in the Appendix.

1 Support Vector Machines: SVMs with Pegasos

In this first problem we will use Support Vector Machines to predict whether the sentiment of a movie review was *positive* or *negative*. We will represent each review by a vector $\boldsymbol{x} \in \mathbb{R}^d$ where d is the size of the word dictionary and x_i is equal to the number of occurrence of the i-th word in the review $\boldsymbol{x}$. The corresponding label is either $y = 1$ for a positive review or $y = -1$ for a negative review. In class we have seen how to transform the SVM training objective into a quadratic program using the dual formulation. Here we will use a gradient descent algorithm instead.

Subgradients

Recall that a vector $g \in \mathbb{R}^d$ is a *subgradient* of $f : \mathbb{R}^d \to \mathbb{R}$ at $\boldsymbol{x}$ if for all $\boldsymbol{z}$,

$$f(\boldsymbol{z}) \geq f(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}).$$

There may be 0, 1, or infinitely many subgradients at any point. The *subdifferential* of f at a point $\boldsymbol{x}$, denoted $\partial f(\boldsymbol{x})$, is the set of all subgradients of f at $\boldsymbol{x}$. A good reference for subgradients are the course notes on Subgradients by Boyd et al. Below we derive a property that will make our life easier for finding a subgradient of the hinge loss.

1. Suppose $f_1, \ldots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_{i=1,\ldots,m} f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $g \in \partial f_k((\boldsymbol{x})$ (a convex function on $\mathbb{R}^d$ has a non-empty subdifferential at all points). Show that $g \in \partial f(\boldsymbol{x})$.

   - we can show this directly
   - as $f_k(x) = f(x)$ and $g \in \partial f(x)$ it must be the case $\forall z \in \mathbb{R}^d \; f_k(z) \geq f_k(x) + g^t(z - x)$
   - this allows us to write $f_k(x) + g^t(z - x) = max(f_1(x)...f_n(x)) + g^t(z - x) \geq f_i(x) + g^t(z - x) \forall i \in [1..m]$ and then by the definition of $f(x)$ we can say $\forall z \in \mathbb{R}^d f(z) = max(f_1(z)...f_m(z)) \geq f_k(z) \geq f_k(x) + g^t(z - x) = max(f_1(x)...f_n(x)) + g^t(z - x) = f(x) + g^t(z - x)$
   - meaning $\forall z \in \mathbb{R}^d f(z) \geq f(x) + g^t(z - x) \Rightarrow g \in \partial f(x)$

2. Give a subgradient of the hinge loss objective $J(\boldsymbol{w}) = \max\{0, 1 - y\boldsymbol{w}^T\boldsymbol{x}\}$.

   - by the above problem we know that a sub gradient of a the max of a combination of convex functions at any given point x is given by a subgradient of the max of those functions.
   - it is clear that $f_1(w) = 0$ is convex as it is constant. it is also clear that $f_2(w) = 1 - y_i w^t x$ is convex in w as it is linear in w .
   - thus the subgradient for $j(w)$ at the point of discontinuity that is when $y_i w^t x_i = 1$, will be given a subgradient of either $f_1(w) = 0$ or $f_2(2) = 1 - y_i w^t x_i$ as at that point $0 = 1 - y_i w^w x_i$ and thus $f_1(w) f_2(w)$
   - further we know that for differentiable convex functions there subgradient are there gradients.
   - so in other words any linear combination of the gradients of $0, 1 - y_i w^t x_i$ will be a subgradient of $j(w)$ when $y_i w^t x_i = 1$
   - so our overall hinge loss subgradient is $\partial J(w) = \begin{cases} 0 & \text{if } y_i w^t x_i > 1 \\ -y_i x_i & \text{if } y_i w^t x_i < 1 \\ \theta(-y_i x_i) & \text{else} \end{cases}$ where $\theta \in [0, 1]$

3. (Optional) Suppose we have function $f : \mathbb{R}^n \to \mathbb{R}$ which is sub-differentiable everywhere, i.e. $\partial f \neq \emptyset$ for all $x \in \mathbb{R}^n$. Show that f is convex. Note, in the general case, a function is convex if for all x, y in the domain of f and for all $\theta \in (0, 1)$,

$$\theta f(a) + (1 - \theta) f(b) \geq f(\theta a + (1 - \theta)(b))$$

   Hint: Suppose f is not convex, then by definition, there exists a point in some interval: $x_0 \in (a, b)$, such that $f(x_0)$ lies above the line connection $(a, f(a)), (b, f(b))$. Is this possible if the function s sub-differentiable everywhere?

   - Sup pose f is not convex, then by definition, there exists a point in some interval: $x_0 \in (a, b)$, such that $f(x_0)$ lies above the line connection $(a, f(a)), (b, f(b))$.
   - we know f has a sub differentiable everywhere that is $\exists \sigma \in \mathbb{R}^d$ such that $\forall z \in \mathbb{R}^d$ $f(z) \geq f(x) + \sigma^t(z - x)$ including x_0
   - we know that x_0 is is inbetween a and b. so then there must exist some combination $\theta a + (1 - \theta)b = x_0$ we know that
   - since x_0 has a subdiferntial it must be the case that $f(\theta a) \geq f(x_0) + \sigma(\theta a - x) \Rightarrow f(\theta a) - \sigma(\theta a - x_0) \geq f(x_0)$ and similary $f((1 - \theta)b) \geq f(x_0) + \sigma((1 - \theta)b - x_0) \Rightarrow f((1 - \theta)b) - \sigma((1 - \theta)b - x) \geq f(x_0)$
   - then for any $\lambda \in [0, 1]$ we can write $\lambda f(\theta a) - \lambda\sigma(\theta a - x_0) + (1 - \lambda) f((1 - \theta)b) - (1 - \lambda)\sigma((1 - \theta)b - x) \geq f(x_0)$
   - this simpflies to $\lambda f(\theta a) + (1 - \lambda) f((1 - \theta)b) + \lambda\theta x_0 + (1 - \lambda)(1 - \theta)(x_0) + \lambda(\theta a) + (1 - \lambda)(1 - \theta b) = \lambda f(\theta a) + (1 - \lambda) f((1 - \theta)b) + x_0 - x_0 = \lambda f(\theta a) + (1 - \lambda) f((1 - \theta)b) \geq f(x_0) = f(\lambda(\theta a) + (1 - \lambda)((1 - \theta)b)$
   - meaning that x_0 is bellow the line conecting a, b and thus the function is convex.

SVM with the Pegasos algorithm

You will train a Support Vector Machine using the Pegasos algorithm [1]. Recall the SVM objective using a linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ and the hinge loss:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i\right\},$$

where n is the number of training examples and d the size of the dictionary. Note that, for simplicity, we are leaving off the bias term b. Note also that we are using ℓ_2 regularization with a parameter λ. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$ for iteration number t. The pseudocode is given below:

Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, n$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t\lambda)w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t\lambda)w_t$

---

4. Consider the SVM objective function for a single training point[2]: $J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\left\{0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i\right\}$. The function $J_i(\boldsymbol{w})$ is not differentiable everywhere. Specify where the gradient of $J_i(w)$ is not defined. Give an expression for the gradient where it is defined.

   - the objective function in this case is not differentiable when $y_i w^t x_i = 1$
   - this makes sense as at that point 0 and $1 - y_i w^t x_i$ have the same value, but there gradients are not equal so the function is not continuous or differentiable at that point.

- when the gradient is defined we have $\nabla J_i(w) = \begin{cases} \lambda w - y_i x_i & \text{if } y_i w^t x_i < 1 \\ \lambda w & \text{if } y_i w^t x_i > 1 \end{cases}$

5. Show that a subgradient of $J_i(w)$ is given by

$$\boldsymbol{gw} = \begin{cases} \lambda\boldsymbol{w} - y_i\boldsymbol{x}_i & \text{for } y_i\boldsymbol{w}^T\boldsymbol{x}_i < 1 \\ \lambda\boldsymbol{w} & \text{for } y_i\boldsymbol{w}^T\boldsymbol{x}_i \geq 1. \end{cases}$$

   You may use the following facts without proof: 1) If $f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(\boldsymbol{x}) = \partial f_1(\boldsymbol{x}) + \cdots + \partial f_n(\boldsymbol{x})$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha\partial f(x)$. (Hint: Use the first part of this problem.)

   - first we want to show that $\frac{\lambda}{2}||w||_2^2$ is convex. we can see that the partial derivative of its gradient is $\lambda > 0$ for all values of w so thus it is convex
   - next we want to show that $max(0, 1-y_i w^t x_i) = max(f_1(w), f_2(w))$ is convex. $f_1''(w) = 0 \geq 0 \forall w$ so f_1 is convex $f_2''(w) = 0 \geq 0$ so this it is also convex. further we know that the max of two convex functions is also convex so thus $max(0, 1 - y_i w^t x_i)$ is convex

[1] Shalev-Shwartz et al. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

[2] Recall that if i is selected uniformly from the set $\{1, \ldots, n\}$, then this objective function has the same expected value as the full SVM objective function.

- so now we can apply the first given fact to say $\partial j_i(w) = \partial \frac{1\lambda 2}{|}|w||_2^2 + \partial max(0, 1 - y_i w^t x_i) = \lambda w + \partial max(0, 1 - y_i w^t x_i)$
- we showed earlier that our overall hinge loss subgradient is $\partial J(w) = \begin{cases} 0 & \text{if } y_i w^t x_i > 1 \\ -y_i x_i & \text{if } y_i w^t x_i < 1 \\ \theta(-y_i x_i) & \text{else} \end{cases}$ where $\theta \in [0, 1]$
- so the formulation given in the problem already holds bassed on that when $y_i w^t x_i \neq 1$
- furhter if we chose $\theta = 1$ we can write using the second fact given in the problem $\partial max(0, 1 - y_i w^t x_i) = \theta(-y_i x_i) = -y_i x_i$
- thus we can see that a valid gradient for $j_i(w)$ is given by

$$\boldsymbol{gw} = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1. \end{cases}$$

- whcih is what we wanted to show

Convince yourself that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous question is the same as given in the pseudocode.

Dataset and sparse representation

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee, provided in the `data_reviews` folder. It has the full text from 2000 movies reviews: 1000 reviews are classified as *positive* and 1000 as *negative*. Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg". We have provided some code in `utils_svm_reviews.py` to assist with reading these files. The code removes some special symbols from the reviews and shuffles the data. Load all the data to have an idea of what it looks like.

A usual method to represent text documents in machine learning is with *bag-of-words*. As hinted above, here every possible word in the dictionnary is a feature, and the value of a word feature for a given text is the number of times that word appears in the text. As most words will not appear in any particular document, many of these counts will be zero. Rather than storing many zeros, we use a *sparse representation*, in which only the nonzero counts are tracked. The counts are stored in a key/value data structure, such as a dictionary in Python. For example, "Harry Potter and Harry Potter II" would be represented as the following Python dict: `x={'Harry':2, 'Potter':2, 'and':1, 'II':1}`.

6. Write a function that converts an example (a list of words) into a sparse bag-of-words representation. You may find Python's Counter[3] class to be useful here. Note that a Counter is itself a dictionary.

```
def bag_of_words(list_of_wrods:list)->dict:
    """
    input: list of strings (or string)
    output: dict containing word counts from list
    """
    return Counter(list_of_wrods)
```

[3] `https://docs.python.org/2/library/collections.html`

7. Load all the data and split it into 1500 training examples and 500 validation examples. Format the training data as a list `X_train` of dictionaries and `y_train` as the list of corresponding 1 or -1 labels. Format the test set similarly.

```
def train_test_split(data: list)->list:
    """"
    input: list of data from read data function
    output: X_train, X_validate: training,validation bag of words
↪   dicts, y_train, y_validate: training and validation labels
    reshuffles data with a set seed for reproducabilty
    """
    rng=np.random.default_rng(15100873)
    rng.shuffle(data)
    train=data[:1500]
    X_train=[bag_of_words(review[:-1]) for review in train]
    y_train=[review[-1] for review in train]
    validate=data[1500:None]
    X_validate=[bag_of_words(review[:-1]) for review in validate]
    y_validate=[review[-1] for review in validate]
    return X_train, y_train, X_validate, y_validate
X_train, y_train, X_validate, y_validate=train_test_split(data)
```

We will be using linear classifiers of the form $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$, and we can store the $\boldsymbol{w}$ vector in a sparse format as well, such as `w={'minimal':1.3, 'Harry':-1.1, 'viable':-4.2, 'and':2.2, 'product':9.1}`. The inner product between $\boldsymbol{w}$ and $\boldsymbol{x}$ would only involve the features that appear in both `x` and `w`, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be `x(Harry) * w(Harry) + x(and) * w(and) = 2*(-1.1) + 1*(2.2)`. To help you along, `utils_svm_reviews.py` includes two functions for working with sparse vectors: 1) a dot product between two vectors represented as dictionaries and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

8. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector $\boldsymbol{w}$ represented as a dictionary. Note that our Pegasos algorithm starts at $w = 0$, which corresponds to an empty dictionary. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch. **Also**: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as 0 should stay at 0.

```
def Pegasos_naive(X_train, y_train,epochs=5000,
↪   regularization_coef=.05):
    """
    input:
    X_train: training data of shape (NxD) (each row is dictionary
↪   storign the word counts of that documnet )
    y_train: training labels of shape (Nx1) (array)
```

```
    epochs: number of epochs to run over (int)
    regularization_coef: lambda value used for normalization (float)
    output:
    w: dictionary of length d with all words and asociated learned
↪   weights

    """
    ## set up
    rng=np.random.default_rng(15100873)
    w_t=dict() ## initilize w
    #rng=np.random.default_rng(15100873) ## initilize random number
    ↪  generator with seed
    curent_epoch=0 ##  epoch counter
    data=list(zip(X_train, y_train)) ## zip data togther to make easy
    ↪  to loop through
    t=0
    while(curent_epoch<=epochs):
        rng.shuffle(data)
        for x_i,y_i in data:
            t+=1
            eta_t=(1)/(regularization_coef*t)
            margin=y_i*dotProduct(w_t, x_i)
            scale=(-1)*eta_t*regularization_coef
            increment(d1=w_t, scale=scale, d2=w_t)
            if margin<1: ## go through each dictionary. if we get the
            ↪  reveiew wrong
                scale=eta_t*y_i
                increment(d1=w_t, scale=scale, d2=x_i) ## we incremnt
                ↪  w_t meaning words in that document are now
                ↪  explicitly included in w_t
        curent_epoch+=1
        print(curent_epoch)
    return w_t
```

Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1 - \eta_t\lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbb{R}$ and $W \in \mathbb{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1 - \eta_t\lambda)\, s_t$.

9. If the update is $w_{t+1} = (1 - \eta_t\lambda)w_t + \eta_t y_j x_j$, then verify that the Pegasos update step is equivalent to:

$$\begin{aligned} s_{t+1} &= (1 - \eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

   Implement the Pegasos algorithm with the (s, W) representation described above.

```
def Pegasos_better(X_train, y_train,epochs=5000,
↪  regularization_coef=.05):
    """
    input:
    X_train: training data of shape (NxD) (each row is dictionary
↪   storign the word counts of that documnet )
    y_train: training labels of shape (Nx1) (array)
    epochs: number of epochs to run over (int)
    regularization_coef: lambda value used for normalization (float)
    output:
    w: dictionary of length d with all words and asociated learned
↪   weights

    """
    ## set up
    rng=np.random.default_rng(15100873)
    W_t=dict() ## which represnts 1/s_t times our independent output
    curent_epoch=0 ##  epoch counter
    data=list(zip(X_train, y_train)) ## zip data togther to make easy
    ↪  to loop through
    t=1
    s_t=1 ## intiilize s_t
    while(curent_epoch<=epochs):
        rng.shuffle(data)
        for x_i,y_i in data:
            t=t+1
            eta_t=(1)/(regularization_coef*t)
            margin=y_i*dotProduct(W_t, x_i)
            a=1-eta_t*regularization_coef
            s_t=(a)*s_t ## s_t updates
            if margin<1:
                scale=(1/s_t)*eta_t*y_i
                increment(d1=W_t, scale=scale, d2=x_i) ##update w_t
        curent_epoch+=1
    w_t={key:W_t[key]*s_t for key in W_t.keys() } ## calulate intended
    ↪  value w_t
    return w_t
```

4

10. Run both implementations of Pegasos on the training data for a couple epochs. Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

[4] There is one subtle issue with the approach described above: if we ever have $1 - \eta_t \lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation.

- my original implementation took around 3 minutes too run 10 epochs
- my optimized implementation took around 1.5 seconds to run 10 epochs

11. Write a function `classification_error` that takes a sparse weight vector `w`, a list of sparse vectors `X` and the corresponding list of labels `y`, and returns the fraction of errors when predicting y_i using $\text{sign}(\boldsymbol{w}^T\boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

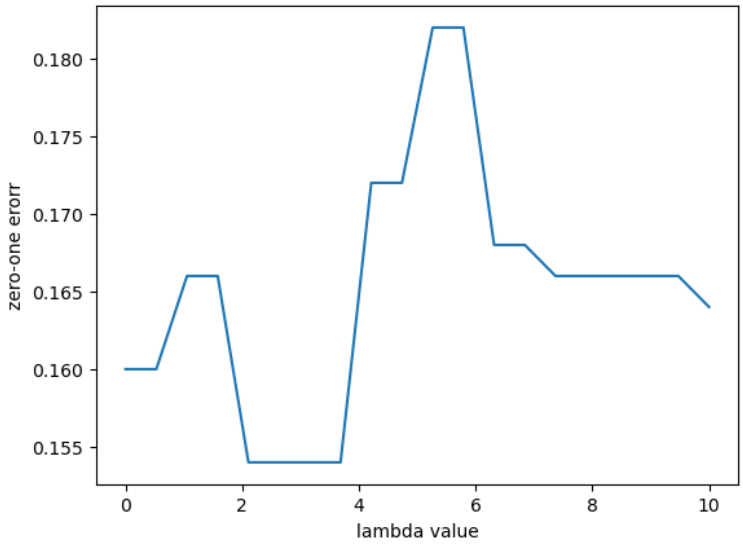
```
        #if key in w_3.keys():
        print(key,w_2[key]==w_3[key])
        print("word: {} \n,naive value={}\n better
        ↪  value={}".format(key,w_2[key], w_3[key]))
```

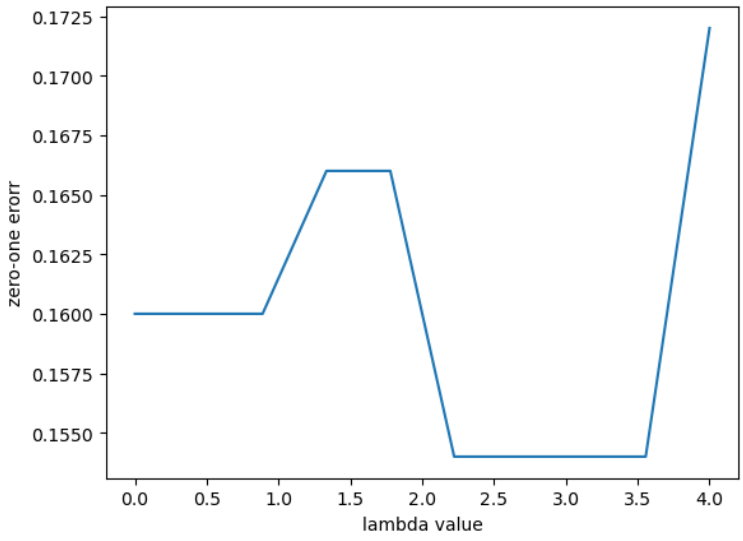
```
# %% [markdown]
# ## question 11

# %%
np.random.default_rng(15100873)
def classification_error(w, X, y):
    """
    input:
    w: wieght dictionary of size d (effectily a DX1 vector)
    X: array of word counts for each document size (nXd)
    y: lables for each document Nx1 array
    output:
    percentage of documents missclassfied using Xw to predict y.
    """
    np.random.default_rng(15100873)
    fixed_dot = lambda x: dotProduct(w,x)
    preds=np.array(list(map(fixed_dot, X)))
    margins=np.array(y)*preds
    return np.sum(margins<0)/len(margins)
```

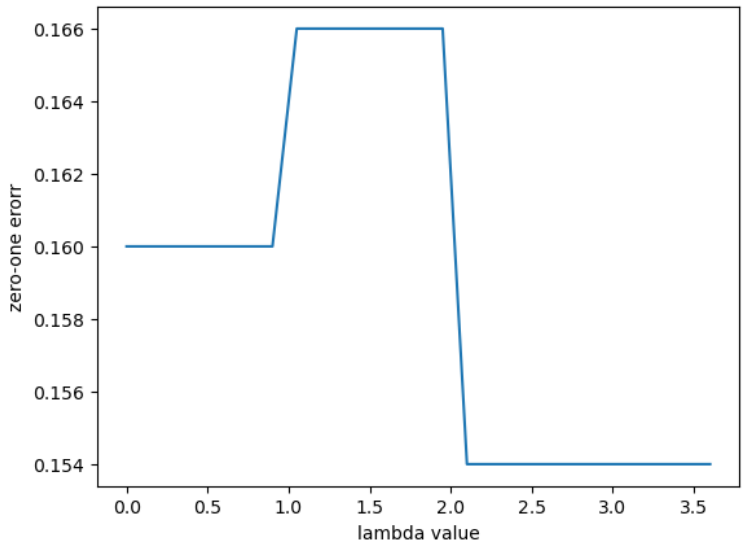
12. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Plot the test errors you obtained as a function of the parameters λ you tested. (Hint: the error you get with the best regularization should be closer to 15% than 20%. If not, maybe you did not train to convergence.)

```
    def test_error_over_alpha(min_alpha, max_alpha,X_train, y_train,
    ↪  X_test, y_test,granulairty=20, epochs=20):
        np.random.default_rng(15100873)
        out=[]
        plt_range=np.linspace(min_alpha, max_alpha,granulairty)
        for alpha in plt_range:
```

```
        w=Pegasos_better(X_train, y_train, epochs=epochs,
        ↪  regularization_coef=alpha)
        out.append(classification_error(w, X_test, y_test))
    plt.plot(plt_range, out)
    plt.xlabel("lambda value")
    plt.ylabel("zero-one erorr")
    plt.show()
```



- 

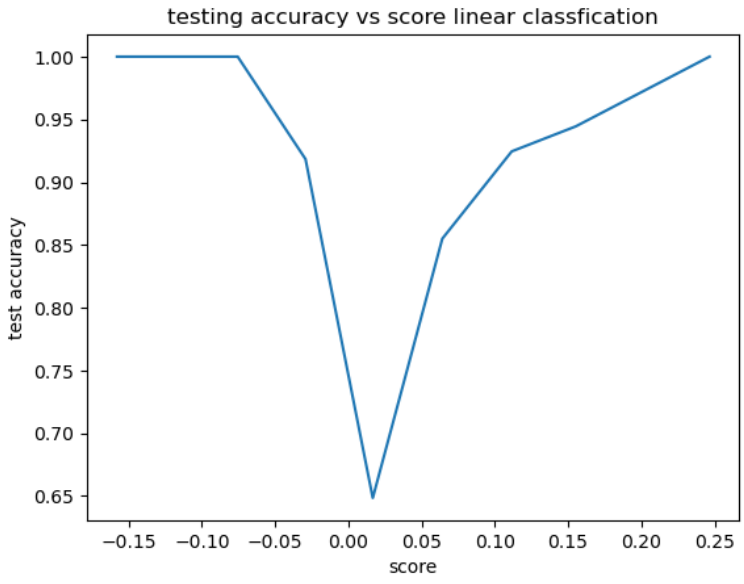


- so i chose my ideal λ to be 2

Error Analysis (Optional)

Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute.

13. (Optional) Break the predictions on the test set into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?



	lower bound	upper bound	test_accuracy
0	-0.213710	-0.157938	1.000000
1	-0.147895	-0.118831	1.000000
2	-0.111254	-0.075547	1.000000
3	-0.075481	-0.029241	0.918367
4	-0.028730	0.016665	0.648485
5	0.016820	0.064126	0.854962
6	0.064548	0.111521	0.924528
7	0.112076	0.155227	0.944444
8	0.157160	0.246573	1.000000

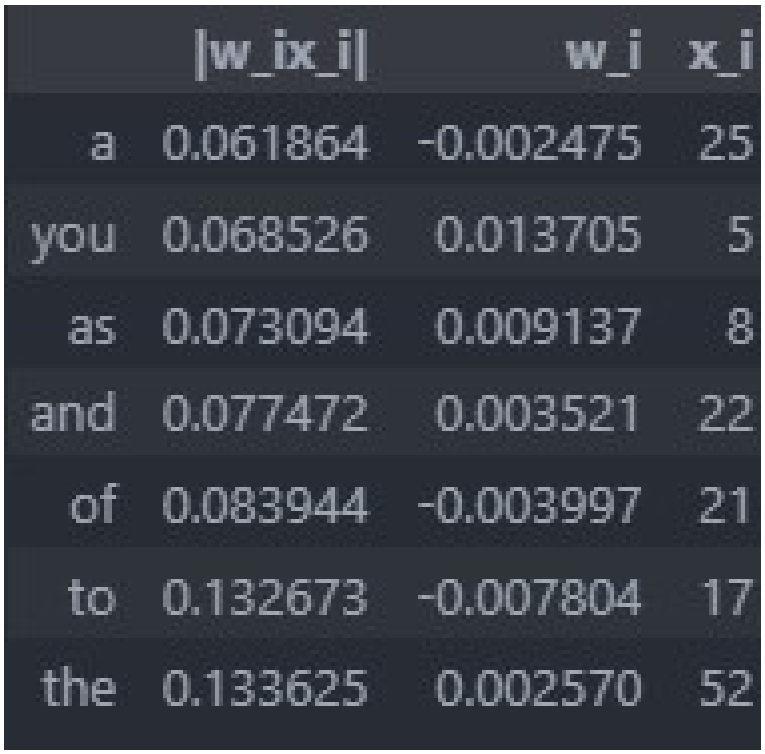
- yes there was a co relation of absolute value 0.79 of magnitude scores and accuracy

In natural language processing one can often interpret why a model has performed well or poorly on a specific example. The first step in this process is to look closely at the errors that the model makes.

14. (Optional) Choose an input example $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ that the model got wrong. We want to investigate what features contributed to this incorrect prediction. One way to rank the importance of the features to the decision is to sort them by the size of their contributions to the score. That is, for each feature we compute $|w_i x_i|$, where w_i is the weight of the ith feature in the prediction function, and x_i is the value of the ith feature in the input x. Create a table of the most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. Attempt to explain why the model was incorrect. Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 2 incorrect examples. Can you think of new features that might help fix a problem? (Think of making groups of words.)

	|w_ix_i|	w_i	x_i
as	0.045684	0.009137	5
and	0.045779	0.003521	13
a	0.051965	-0.002475	21
of	0.055963	-0.003997	14
look	0.077948	-0.008661	9
to	0.078043	-0.007804	10
the	0.136195	0.002570	53

	|w_ix_i|	w_i	x_i
also	0.037118	0.012373	3
have	0.044827	-0.014942	3
two	0.046160	-0.009232	5
of	0.059960	-0.003997	15
and	0.063386	0.003521	18
the	0.115637	0.002570	45
to	0.117065	-0.007804	15



	|w_ix_i|	w_i	x_i
a	0.061864	-0.002475	25
you	0.068526	0.013705	5
as	0.073094	0.009137	8
and	0.077472	0.003521	22
of	0.083944	-0.003997	21
to	0.132673	-0.007804	17
the	0.133625	0.002570	52

- for these three examples all the words are very ambiguous. it makes sense that in isolation it is unclear how words like, "as" or "and" should impact classification. Perhaps analyzing these texts with a larger unit of analysis like sentences or groups of words as opposed to single words.

2 Kernel Methods

2.1 Kernelization review

Consider the following optimization problem on a data set $(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathcal{Y}$:

$$\min_{w \in \mathbb{R}^d} R\left(\sqrt{\langle \boldsymbol{w}, \boldsymbol{w} \rangle}\right) + L\left(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle\right),$$

where $\boldsymbol{w}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$, and $\langle \cdot, \cdot \rangle$ is the standard inner product on $\mathbb{R}^d$. The function $R : [0, \infty) \to \mathbb{R}$ is nondecreasing and gives us our regularization term, while $L : \mathbb{R}^n \to \mathbb{R}$ is arbitrary[5] and gives us our loss term. We noted in lecture that this general form includes soft-margin SVM and ridge regression, though not lasso regression. Using the representer theorem, we showed if the optimization problem has a solution, there is always a solution of the form $\boldsymbol{w} = \sum_{i=1}^n \boldsymbol{\alpha}_i \boldsymbol{x}_i$, for some $\alpha \in \mathbb{R}^n$. Plugging this into the our original problem, we get the following "kernelized" optimization problem:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} R\left(\sqrt{\boldsymbol{\alpha}^T K \boldsymbol{\alpha}}\right) + L\left(K\boldsymbol{\alpha}\right),$$

[5] You may be wondering "Where are the y_i's?". They're built into the function L. For example, a square loss on a training set of size 3 could be represented as $L(s_1, s_2, s_3) = \frac{1}{3}\left[(s_1 - y_1)^2 + (s_2 - y_2)^2 + (s_3 - y_3)^3\right]$, where each s_i stands for the ith prediction $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

where $K \in \mathbb{R}^{n \times n}$ is the Gram matrix (or "kernel matrix") defined by $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Predictions are given by

$$f(x) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}),$$

and we can recover the original $\boldsymbol{w} \in \mathbb{R}^d$ by $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$.

The *kernel trick* is to swap out occurrences of the kernel k (and the corresponding Gram matrix K) with another kernel. For example, we could replace $k(x_i, x_j) = \langle x_i, x_j \rangle$ by $k'(x_i, x_j) = \langle \psi(x_i), \psi(x_j) \rangle$ for an arbitrary feature mapping $\boldsymbol{\psi} : \mathbb{R}^d \to \mathbb{R}^d$. In this case, the recovered $\boldsymbol{w} \in \mathbb{R}^d$ would be $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{\psi}(\boldsymbol{x}_i)$ and predictions would be $\langle \boldsymbol{w}, \boldsymbol{\psi}(\boldsymbol{x}_i) \rangle$.

More interestingly, we can replace k by another kernel $k''(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for which we do not even know or cannot explicitly write down a corresponding feature map $\boldsymbol{\psi}$. Our main example of this is the RBF kernel

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right),$$

for which the corresponding feature map ψ is infinite dimensional. In this case, we cannot recover w since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbb{R}^n$, with $f(x) = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$.

Your implementation of kernelized methods below should not make any reference to $\boldsymbol{w}$ or to a feature map $\boldsymbol{\psi}$. Your learning routine should return $\boldsymbol{\alpha}$, rather than $\boldsymbol{w}$, and your prediction function should also use $\boldsymbol{\alpha}$ rather than $\boldsymbol{w}$. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

2.2 Kernel problems

Ridge Regression: Theory

Suppose our input space is $\mathcal{X}$=$\mathbb{R}^d$ and our output space is $\mathcal{Y} = \mathbb{R}$. Let $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ be a training set from $\mathcal{X} \times \mathcal{Y}$. We'll use the "design matrix" $X \in \mathbb{R}^{n \times d}$, which has the input vectors as rows:

$$X = \begin{pmatrix} -\boldsymbol{x}_1 - \\ \vdots \\ -\boldsymbol{x}_n - \end{pmatrix}.$$

Recall the ridge regression objective function:

$$J(\boldsymbol{w}) = ||X\boldsymbol{w} - y||^2 + \lambda ||\boldsymbol{w}||^2,$$

for $\lambda > 0$.

15. Show that for $\boldsymbol{w}$ to be a minimizer of $J(\boldsymbol{w})$, we must have $X^T X \boldsymbol{w} + \lambda I \boldsymbol{w} = X^T y$. Show that the minimizer of $J(\boldsymbol{w})$ is $\boldsymbol{w} = (X^T X + \lambda I)^{-1} X^T y$. Justify that the matrix $X^T X + \lambda I$ is invertible, for $\lambda > 0$. (You should use properties of positive (semi)definite matrices. If you need a reminder look up the Appendix.)

- $J(W) = ||Xw - y||_2^2 + \lambda ||w||_2^2 = (w^w x^t X w - w^t X^t y - y^t x w)^2 + (\lambda w^t W)^2$
- we can take the gradient of this to see that $\nabla_j(w) = 4X^T X w - 4X^T y + 4\lambda w$ item setting this equal to zero and solving yields $w^* = (X^T X + \lambda I)^{-1}(X^t y)$ as the only extrema of our function so if there is a min it will be at that point

- taking the second derivative yields $4\lambda \geq 0$ so the function is convex,and thus we know $J(W)$ will have one unique minimum at $w^* = (X^T X + \lambda I)^{-1}(X^t y)$
- next we want to show $(X^T X + \lambda I)$ is invertable
  - to show a matrix is invertable we just need to show it has no zero eigenvalues
  - a matrix A is positive definite if $\forall v \in \mathbb{R}^d \; v^t A v > 0$
  - so for an arbitrary vector v we can write $v^t(X^T X + \lambda I)v = v^t X^T X v + v^T \lambda I V$
  - we can see that $v^T(X^T X)v = v^t \Sigma_{i=1}^d (x_i)^t X_i v = \Sigma_{j=1}^d v_j \Sigma_{i=1}^n (X_{i,j}^t X_{i,j}) v_j = \Sigma_{i=1}^d \Sigma_{j=1}^d (X_{i,j})^2 (V_i)^2 \geq 0$ since it is the sum of squared quantities
  - $v^T \lambda I v = \lambda v^T I v = \lambda v^t v = \lambda ||v||$ for any nonzero vector v $||v|| > 0$ and $\lambda > 0$ thus $v^t \lambda I v > 0$
  - so for an arbitrary vector v we can write $v^t(X^T X + \lambda I)v = v^t X^T X v + v^T \lambda I v > 0$
  - meaning $v^t(X^T X + \lambda I)v$ is positive definite and thus has no zero eigenvalues
  - meaning that $v^t(X^T X + \lambda I)v$ is invertable

16. Rewrite $X^T X w + \lambda I w = X^T y$ as $\boldsymbol{w} = \frac{1}{\lambda}(X^T y - X^T X w)$. Based on this, show that we can write $w = X^T \boldsymbol{\alpha}$ for some $\boldsymbol{\alpha}$, and give an expression for $\boldsymbol{\alpha}$.

- $X^T X w + \lambda I w = X^t y \Rightarrow \lambda I w = X^t y - X^T X w \Rightarrow I w = \frac{1}{\lambda}(X^t y - X^T X w)$
  $\Rightarrow w = \frac{1}{\lambda}(X^t y - X^T X w)$
- we can write $w = \frac{1}{\lambda}(X^t y - X^T X w)$ as $\frac{1}{\lambda}(X^t y - X^T X w) = X^t \frac{1}{\lambda}(y - X w)$ and then define $\frac{1}{\lambda}(y - X w) = \alpha \in \mathbb{R}^n$ thus we have $w = X^T \alpha$ which was what we waned to show

17. Based on the fact that $\boldsymbol{w} = X^T \boldsymbol{\alpha}$, explain why we say $\boldsymbol{w}$ is "in the span of the data."

- for a matrix $v \in \mathbb{R}^{n \times d}$ the span of v is def fined as $span(v) = span(v_1 ... v_n) = \{v_1 \alpha_1 + v_2 \alpha_2 + ... v_n \alpha_n | \forall \alpha_i \in \mathbb{R}\} = \{\Sigma_{i=1}^n v_i \alpha_i | \forall \alpha_i \in \mathbb{R}\} = \{X^t \alpha | \forall \alpha \in \mathbb{R}^n\}$ that is all linear combinations of its rows.
- as we know X is our data matrix and $w = X^T \alpha$ it must be the case that $w \in span(X) = span(x_1 ... x_n)$ that is we can get w as a linear combination of the rows of our data matrix which represent out data points.

18. Show that $\boldsymbol{\alpha} = (\lambda I + X X^T)^{-1} y$. Note that $X X^T$ is the kernel matrix for the standard vector dot product. (Hint: Replace $\boldsymbol{w}$ by $X^T \boldsymbol{\alpha}$ in the expression for $\boldsymbol{\alpha}$, and then solve for $\boldsymbol{\alpha}$.)

- from question 16 we have $\alpha = \frac{1}{\lambda} y - \frac{1}{\lambda} X w$ adn given $w = X^T \alpha$ we can write $\alpha = \frac{1}{\lambda} y - \frac{1}{\lambda} X^t \alpha \Rightarrow \frac{1}{\lambda} y = (I + \frac{1}{\lambda} X X^t)\alpha \Rightarrow$
  $\frac{1}{\lambda} y (I + \frac{1}{\lambda} X X^t)^{-1} = \alpha \Rightarrow y(\lambda I + X X^t)^{-1} = \alpha$

19. Give a kernelized expression for the $X\boldsymbol{w}$, the predicted values on the training points. (Hint: Replace $\boldsymbol{w}$ by $X^T \boldsymbol{\alpha}$ and $\boldsymbol{\alpha}$ by its expression in terms of the kernel matrix $X X^T$.)

- we can write our prediction as $Xw = X(X^T\alpha) = (XX^T)\alpha = \begin{pmatrix} < x_1, x_1 > & \dots & < x_1, x_n > \\ \dots & \dots & \dots \\ < x_n, x_1 > & \dots & < x_n, x_n > \end{pmatrix} \alpha = K\alpha$

20. Give an expression for the prediction $f(x) = x^T \boldsymbol{w}^*$ for a new point x, not in the training set. The expression should only involve x via inner products with other $\boldsymbol{x}$'s. (Hint: It is often convenient to define the column vector

$$k_{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x}^T \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{x}_n \end{pmatrix}$$

to simplify the expression.)

- $\hat{f}(x) = w^t x =< w^*.x >=< \Sigma_{i=1}^n \alpha_i^* x_i, x >= \Sigma_{i=1}^n < \alpha_i^* x_i, x >= \Sigma_{i=1}^n \alpha_i^* < x_i, x >= k_x^t \alpha^*$

Kernels and Kernel Machines

There are many different families of kernels. So far we spoken about linear kernels, RBF/Gaussian kernels, and polynomial kernels. The last two kernel types have parameters. In this section, we'll implement these kernels in a way that will be convenient for implementing our kernelized ridge regression later on. For simplicity, we will assume that our input space is $\mathcal{X} = \mathbb{R}$. This allows us to represent a collection of n inputs in a matrix $X \in \mathbb{R}^{n \times 1}$. You should now refer to the jupyter notebook `skeleton_code_kernels.ipynb`.

21. Write functions that compute the RBF kernel $k_{\text{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\text{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\text{linear}}(x, x') = \langle x, x' \rangle$, has been done for you in the support code. Your functions should take as input two matrices $W \in \mathbb{R}^{n_1 \times d}$ and $X \in \mathbb{R}^{n_2 \times d}$ and should return a matrix $M \in \mathbb{R}^{n_1 \times n_2}$ where $M_{ij} = k(W_{i\cdot}, X_{j\cdot})$. In words, the (i, j)'th entry of M should be kernel evaluation between w_i (the ith row of W) and x_j (the jth row of X). For the RBF kernel, you may use the scipy function `cdist(X1,X2,'sqeuclidean')` in the package `scipy.spatial.distance`.

```
    """
    Computes the RBF kernel between two sets of vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        sigma - the bandwidth (i.e. standard deviation) for the
↪   RBF/Gaussian kernel
    Returns:
        matrix of size n1xn2, with exp(-||x1_i-x2_j||^2/(2 sigma^2))
↪   in position i,j
    """
    a=cdist(X1,X2,'sqeuclidean')*(-1)/(2*sigma**2)
    return np.exp(a)
    #TODO
```

```
def polynomial_kernel(X1, X2, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of
↪   vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        offset, degree - two parameters for the kernel
    Returns:
        matrix of size n1xn2, with (offset + <x1_i,x2_j>)^degree in
↪   position i,j
    """
    #TODO
    return np.power((offset+np.dot(X1,np.transpose(X2))), degree)
```

22. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. Include both the code and the output.

```
x=np.array([-4,-1,0,2]).reshape(4,1)
linear_kernel(x,x)
```

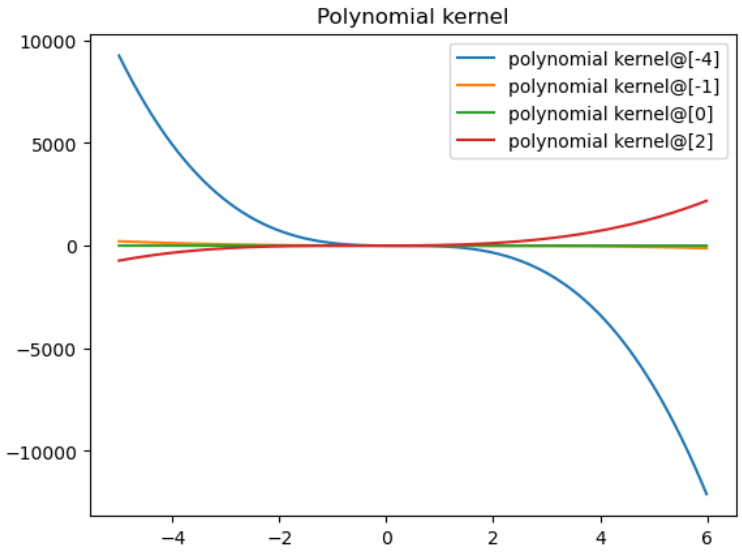
- here is the output
-
```
array([[16,  4,  0, -8],
       [ 4,  1,  0, -2],
       [ 0,  0,  0,  0],
       [-8, -2,  0,  4]])
```

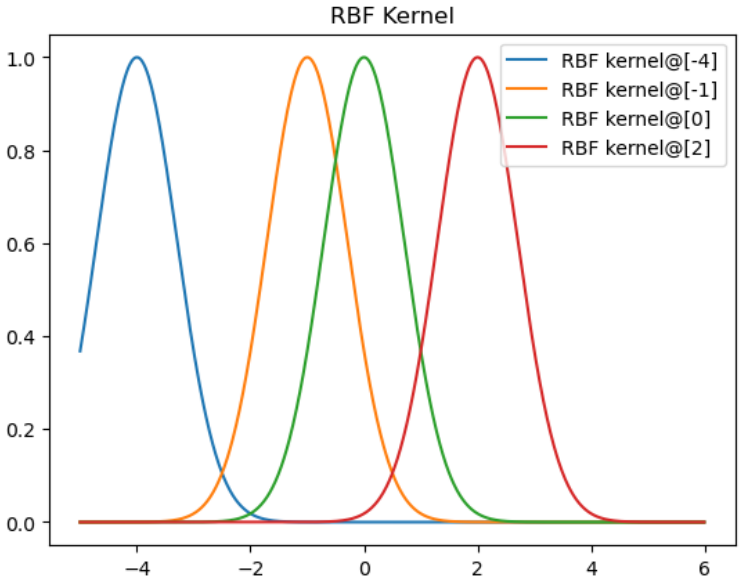
23. Suppose we have the data set $\mathcal{D}_{X,y} = \{(-4, 2), (-1, 0), (0, 3), (2, 5)\}$ (in each set of parentheses, the first number is the value of x_i and the second number the corresponding value of the target y_i). Then by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. This set of functions will look quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6, 6]$ has been provided for the linear kernel.

    (a) Plot the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

    (b) Plot the set of functions $x \mapsto k_{\text{RBF}(1)}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6, 6]$.

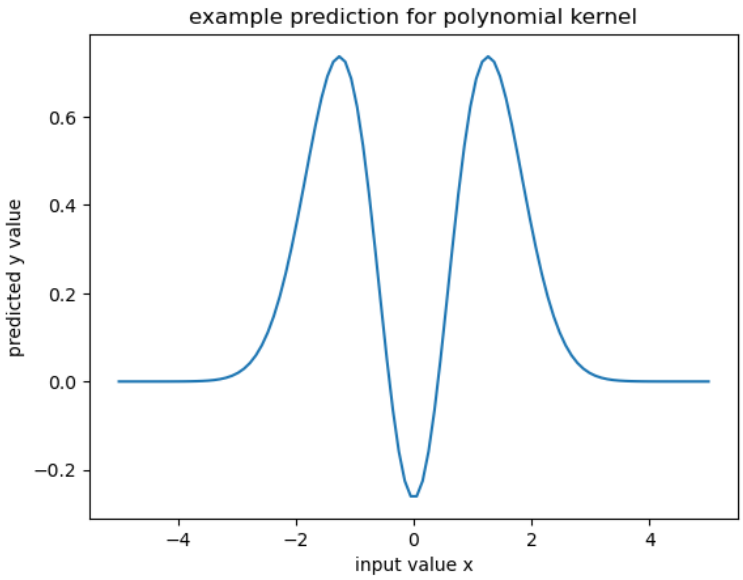
    Note that the values of the parameters of the kernels you should use are given in their definitions in (a) and (b).





24. By the representer theorem, the final prediction function will be of the form $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $x_1, \ldots, x_n \in \mathcal{X}$ are the inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make prediction with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel (sigma=1), with prototype points at $-1, 0, 1$ and corresponding weights α_i $1, -1, 1$. Plot the resulting function.

- here is the plot of

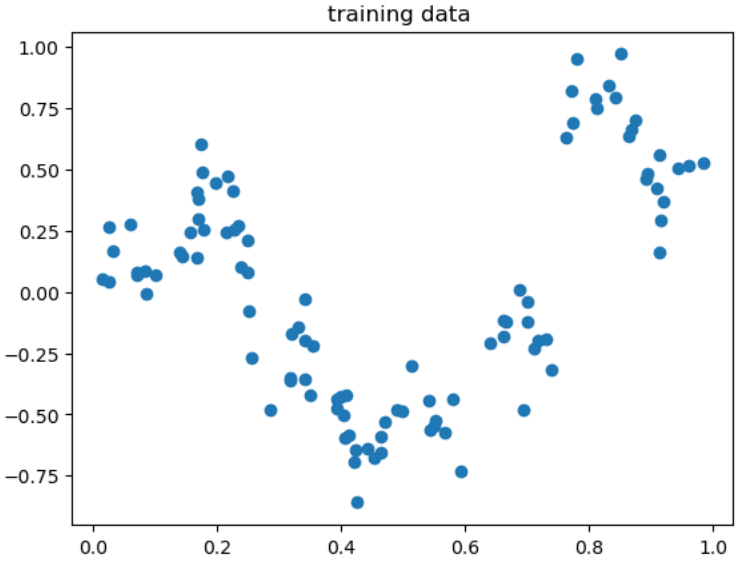


Note: For this last problem, and for other problems below, it may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`, you can write `k = functools.partial(polynomial_kernel, offset=2, degree=2)`, and then a call to `k(W,X)` is equivalent to `polynomial_kernel(W, X, offset=2, degree=2)`, the advantage being that the extra parameter settings are built into `k(W,X)`. This can be convenient so that you can have a function that just takes a kernel function `k(W,X)` and doesn't have to worry about the parameter settings for the kernel.

Kernel Ridge Regression: Practice

In the zip file for this assignment, we provide a training `krr-train.txt` and test set `krr-test.txt` for a one-dimensional regression problem, in which $\mathcal{X} = \mathcal{Y} = \mathcal{A} = \mathbb{R}$. Fitting this data using kernelized ridge regression, we will compare the results using several different kernel functions. Because the input space is one-dimensional, we can easily visualize the results.

25. Plot the training data. You should note that while there is a clear relationship between x and y, the relationship is not linear.

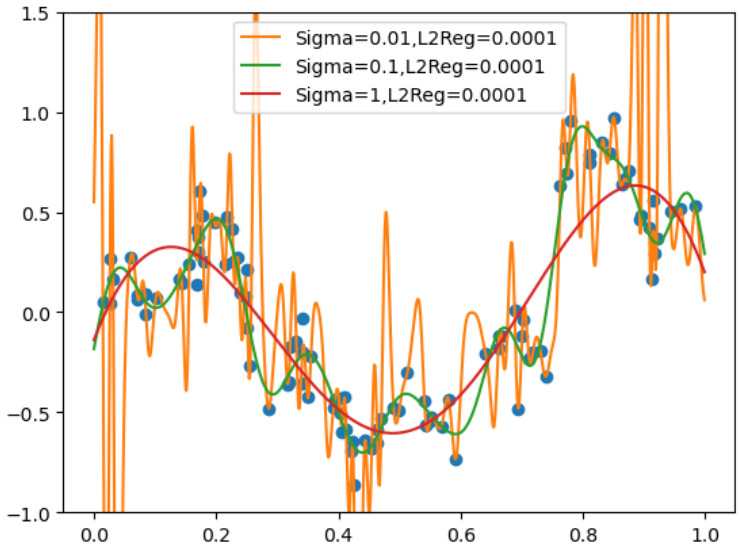


26. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$, where $\alpha = (\lambda I + K)^{-1} y$ and $K \in \mathbb{R}^{n \times n}$ is the kernel matrix of the training data: $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the training point $\boldsymbol{x}_i$. Complete the function `train_kernel_ridge_regression` so that it performs kernel ridge regression and returns a `Kernel_Machine` object that can be used for predicting on new points.

```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        """
        Args:
            kernel(X1,X2) - a function return the cross-kernel matrix
↪   between rows of X1 and rows of X2 for kernel k
            training_points - an nxd matrix with rows x_1,..., x_n
            weights - a vector of length n with entries
↪   alpha_1,...,alpha_n
        """

        self.kernel = kernel
        self.training_points = training_points
        self.weights = weights

    def predict(self, X):
        """
        Evaluates the kernel machine on the points given by the rows
↪   of X
        Args:
            X - an nxd matrix with inputs x_1,...,x_n in the rows
        Returns:
```

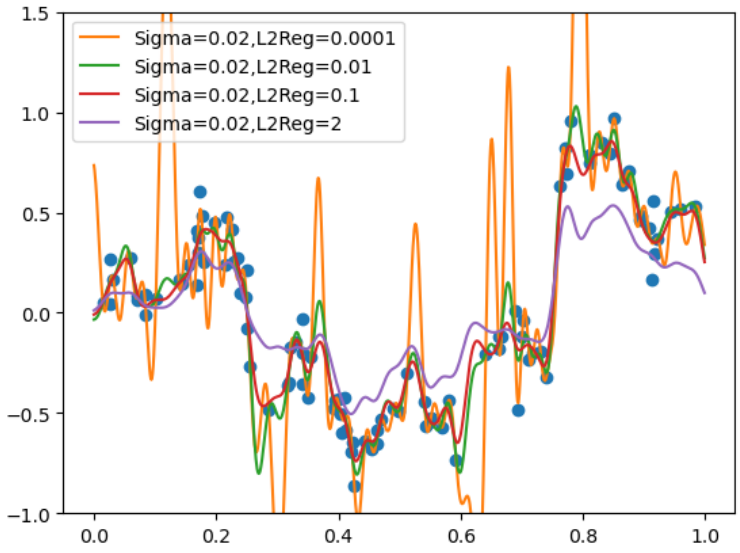
```
            Vector of kernel machine evaluations on the n points in X.
↪  Specifically, jth entry of return vector is
                Sum_{i=1}^R alpha_i k(x_j, mu_i)
        """
        # TODO
        k_t=self.kernel(self.training_points,X)
        return np.dot(k_t.T, self.weights)
```

27. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed regularization parameter of 0.0001 for 3 different values of sigma: 0.01, 0.1, and 1.0. What values of sigma do you think would be more likely to over fit, and which less?



-
- it seems that low values of sigma are very sensitive to changes in the training data so they are more likely to over-fit than lower values of sigma

28. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed sigma of 0.02 and 4 different values of the regularization parameter λ: 0.0001, 0.01, 0.1, and 2.0. What happens to the prediction function as $\lambda \to \infty$?

- 

- it seems that low values of λ are very sensitive to changes in the training data so they are more likely to over-fit while it looks like very high values of λ lead to under fitting to the training data

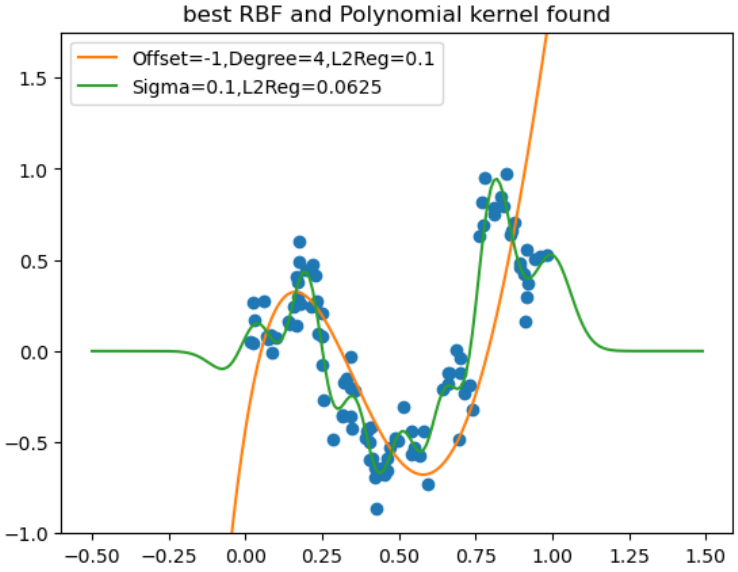
29. (Optional) Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use average square loss on the test set to rank the parameter settings. To make things easier for you, we have provided an sklearn wrapper for the kernel ridge regression function we have created so that you can use sklearn's GridSearchCV. Note: Because of the small dataset size, these models can be fit extremely fast, so there is no excuse for not doing extensive hyperparameter tuning.

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
27	-	RBF	0.0625	-	0.1	0.013866	0.014689
24	-	RBF	0.1250	-	0.1	0.014398	0.015934
21	-	RBF	0.2500	-	0.1	0.015807	0.017876
18	-	RBF	0.5000	-	0.1	0.018022	0.020495
15	-	RBF	1.0000	-	0.1	0.020777	0.023856

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
58	-	linear	1.00	-	-	0.164540	0.206506
59	-	linear	0.01	-	-	0.164569	0.206501
57	-	linear	10.00	-	-	0.164591	0.206780

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
54	4	polynomial	0.01	-1	-	0.043454	0.060135
56	4	polynomial	0.01	1	-	0.060262	0.088844
33	2	polynomial	0.10	-1	-	0.065554	0.098913
38	2	polynomial	0.01	1	-	0.066532	0.097785
36	2	polynomial	0.01	-1	-	0.066915	0.097706

30. (Optional) Plot your best fitting prediction functions using the polynomial kernel and the RBF kernel. Use the domain $x \in (-0.5, 1.5)$. Comment on the results.



-
- both kernel ls seem to capture the data really well.
- it is possible that the polynomial kernel is better as it is fitting the general shape with out getting thrown off by points that might be noise

31. (Optional) The data for this problem was generated as follows: A function $f : \mathbb{R} \to \mathbb{R}$ was chosen. Then to generate a point (x, y), we sampled x uniformly from $(0, 1)$ and we sampled $\epsilon \sim \mathcal{N}\left(0, 0.1^2\right)$ (so $var(\epsilon) = 0.1^2$). The final point is $(x, f(x) + \epsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $\ell\left(\hat{y}, y\right) = (\hat{y} - y)^2$.

- the Bayes function is defined as $f^* : \mathcal{X} \Rightarrow \mathcal{A} : f^* \in argmin_f R(F)$
- in this context the risk of a function g can be expressed as $R(g) = E_{(x,f(x)+\epsilon)\sim P(x\times y)[\ell(g(x),y)]=E_{(x,f(x)+\epsilon)\sim P(x\times y)[(g(x)-f(x)+\epsilon)^2]}$
- ϵ is a random quantity so we can not really minimize the risk around it, thus our Bayes function will be $f^*(x) = f(x)$
- thus we can calculate our Bayes risk as $R(f^*) = E_{(x,f(x)+\epsilon)\sim P(x\times y)}[(f^*(x) - f(x) + \epsilon)^2] = E_{(x,f(x)+\epsilon)\sim P(x\times y)}[(f(x) - f(x) + \epsilon)^2] = E_{(x,f(x)+\epsilon)\sim P(x\times y)}[(\epsilon)^2] = var(\epsilon) = .1^2$
- this makes sense as even if we fully capture the deterministic function we are trying to understand ϵ is still present as an irreducible level of noise

3 Kernel SVMs with Kernelized Pegasos (Optional)

32. (Optional) Load the SVM training `svm-train.txt` and `svm-test.txt` test data from the zip file. Plot the training data using the code supplied. Are the data linearly separable? Quadratically separable? What if we used some RBF kernel?

- the data looks like there is a Circe of one class so rounding the other class

- No the data is not linearly seperable
- since they are thus ceperated by what looks aproxently like a circle it is likely that either an rbh or quadratic kernel should be able to seperate the data.

33. (Optional) Unlike for kernel ridge regression, there is no closed-form solution for SVM classification (kernelized or not). Implement kernelized Pegasos. Because we are not using a sparse representation for this data, you will probably not see much gain by implementing the "optimized" versions described in the problems above.

```
def polynomial_kernel(X1, X2, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of
↪   vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        offset, degree - two parameters for the kernel
    Returns:
        matrix of size n1xn2, with (offset + <x1_i,x2_j>)^degree in
↪   position i,j
    """
    #TODO
    return np.power((offset+np.dot(X1,np.transpose(X2))), degree)

def train_soft_svm(X_train, y_train,kernel,epochs=100,
↪   regularization_coef=.05):
    ## set up
    W_t=np.zeros((X_train.shape[0],1)) ## which represnts 1/s_t our
    ↪   independent output
    seed(15100873) ## set random seed
    curent_epoch=0 ##  epoch counter
    data=list(zip(X_train, y_train)) ## zip data togther to make easy
    ↪   to loop through
    k=kernel(X_train, X_train)
    t=1
    s_t=1 ## intiilize s_5
    while(curent_epoch<=epochs):
        shuffle(data)
        for x_i,y_i in data:
            x_i=x_i.reshape(-1,1)
            t+=1
            eta_t=(1)/(regularization_coef*t)
            a=1-eta_t*regularization_coef
            s_t=(a)*s_t ## s_t updates
            k_t=kernel(X_train,x_i.T)
            pred=np.dot(k_t.T, W_t)
            margin=y_i*pred

            if margin<1:
```

```
                scale=(1/s_t)*eta_t*y_i
                W_t=np.add(W_t,scale*(np.dot( k.T,y_train)))

        curent_epoch+=1

    return Kernel_Machine(kernel, X_train, W_t*s_t)


from sklearn.base import BaseEstimator, RegressorMixin,
↪  ClassifierMixin

class soft_svm(BaseEstimator, RegressorMixin):
    """sklearn wrapper for our kernel ridge regression"""

    def __init__(self, kernel="RBF", sigma=1, degree=2, offset=1,
    ↪  l2reg=1):
        self.kernel = kernel
        self.sigma = sigma
        self.degree = degree
        self.offset = offset
        self.l2reg = l2reg

    def fit(self, X, y=None):
        """
        This should fit classifier. All the "work" should be done
↪   here.
        """
        if (self.kernel == "linear"):
            self.k = linear_kernel
        elif (self.kernel == "RBF"):
            self.k = functools.partial(RBF_kernel, sigma=self.sigma)
        elif (self.kernel == "polynomial"):
            self.k = functools.partial(polynomial_kernel,
            ↪  offset=self.offset, degree=self.degree)
        else:
            raise ValueError('Unrecognized kernel type requested.')

        self.kernel_machine_ = train_soft_svm(X, y, self.k,
        ↪  self.l2reg)

        return self

    def predict(self, X, y=None):
        try:
            getattr(self, "kernel_machine_")
        except AttributeError:
            raise RuntimeError("You must train classifer before
            ↪  predicting data!")
```

```
        return(self.kernel_machine_.predict(X))

    def score(self, X, y=None):
        # get the average square error
        return(((self.predict(X)-y)**2).mean())
```

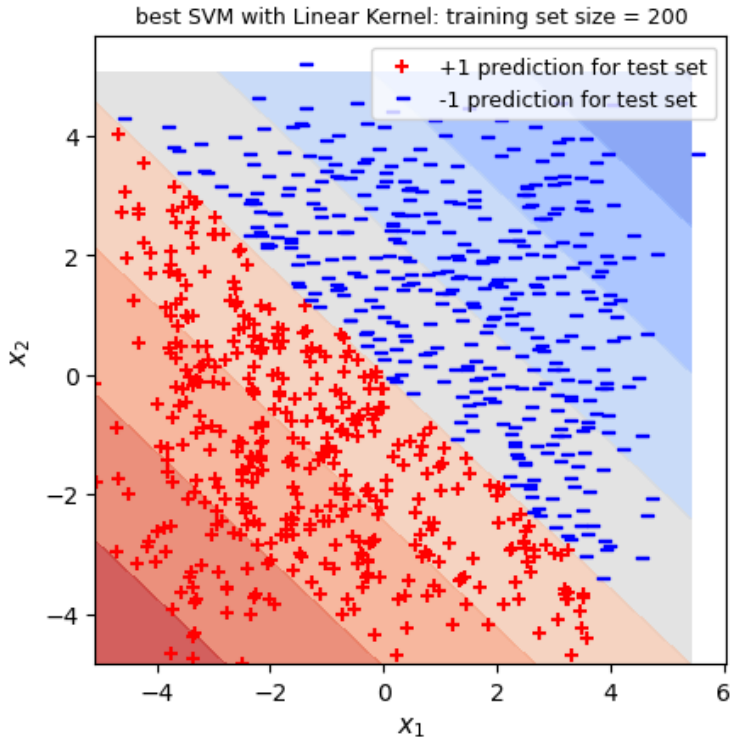
34. (Optional) Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error (i.e. average 0/1 loss) for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use the 0/1 loss on the test set to rank the parameter settings.

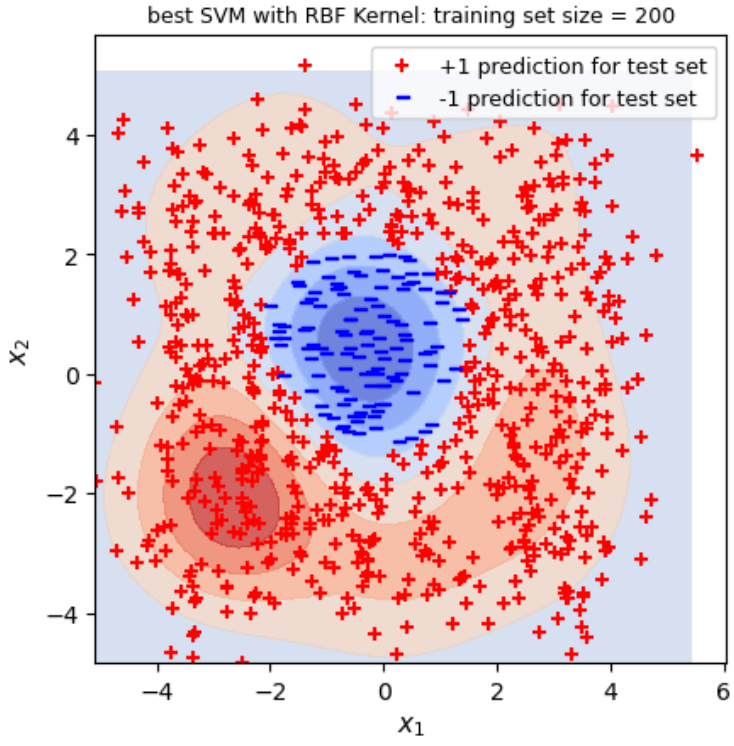
	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
28	-	RBF	0.0625	-	1	0.398537	0.586172
25	-	RBF	0.1250	-	1	0.398537	0.586172
22	-	RBF	0.2500	-	1	0.398537	0.586172
19	-	RBF	0.5000	-	1	0.398537	0.586172
16	-	RBF	1.0000	-	1	0.478049	0.758229

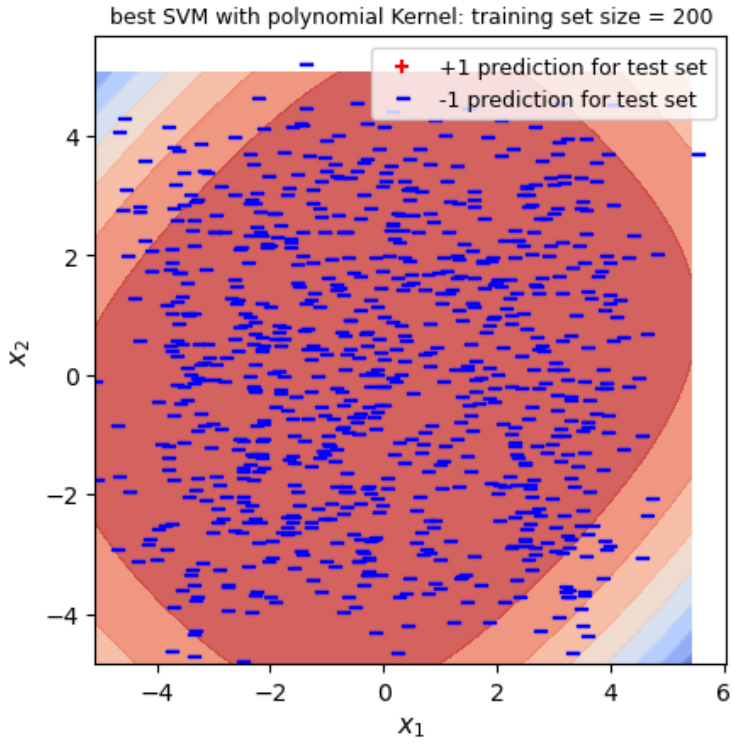
	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
59	-	linear	0.01	-	-	42051.042639	53904.718285
58	-	linear	1.00	-	-	82484.314408	105735.726251
57	-	linear	10.00	-	-	446383.760326	572214.797948

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
48	4	polynomial	10.00	-1	-	9.984580e-01	9.983748e-01
49	4	polynomial	10.00	0	-	9.984873e-01	9.983887e-01
50	4	polynomial	10.00	1	-	9.984966e-01	9.983706e-01
47	3	polynomial	0.01	1	-	9.999999e-01	9.999999e-01
44	3	polynomial	0.10	1	-	9.999999e-01	9.999999e-01

35. (Optional) Plot your best fitting prediction functions using the linear, polynomial, and the RBF kernel. The code provided may help.


best SVM with Linear Kernel: training set size = 200
+1 prediction for test set
-1 prediction for test set
x_1
x_2


best SVM with RBF Kernel: training set size = 200
+1 prediction for test set
-1 prediction for test set
x_1
x_2


best SVM with polynomial Kernel: training set size = 200
+1 prediction for test set
-1 prediction for test set
4
2
0
−2
−4
−4
−2
0
2
4
6
x_1
x_2

Appendix (Not for credit)

Here we are recalling important properties of positive (semi)definite matrices. The exercises below are for revisions for student who may not feel comfortable with these notions. **None of the appendix is for credit.**

A Positive Semidefinite Matrices

In statistics and machine learning, we use positive semidefinite matrices a lot. Let's recall some definitions from linear algebra that will be useful here:

Definition. A set of vectors $\{x_1, \ldots, x_n\}$ is **orthonormal** if $\langle x_i, x_i \rangle = 1$ for any $i \in \{1, \ldots, n\}$ (i.e. x_i has unit norm), and for any $i, j \in \{1, \ldots, n\}$ with $i \neq j$ we have $\langle x_i, x_j \rangle = 0$ (i.e. x_i and x_j are orthogonal).

Note that if the vectors are column vectors in a Euclidean space, we can write this as $x_i^T x_j = \mathbb{1} i \neq j$ for all $i, j \in \{1, \ldots, n\}$.

Definition. A matrix is **orthogonal** if it is a square matrix with orthonormal columns.

It follows from the definition that if a matrix $M \in \mathbb{R}^{n \times n}$ is orthogonal, then $M^T M = I$, where I is the $n \times n$ identity matrix. Thus $M^T = M^{-1}$, and so $MM^T = I$ as well.

Definition. A matrix M is **symmetric** if $M = M^T$.

Definition. For a square matrix M, if $Mv = \lambda v$ for some column vector v and scalar λ, then v is called an **eigenvector** of M and λ is the corresponding **eigenvalue**.

Theorem. [Spectral Theorem]A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ can be diagonalized as $M = Q \Sigma Q^T$, where $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are a set of orthonormal eigenvectors of M, and Σ is a diagonal matrix of the corresponding eigenvalues.

Definition. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is **positive semidefinite (psd)** if for any $x \in \mathbb{R}^n$,

$$x^T M x \geq 0.$$

Note that unless otherwise specified, when a matrix is described as positive semidefinite, we are implicitly assuming it is real and symmetric (or complex and Hermitian in certain contexts, though not here).

As an exercise in matrix multiplication, note that for any matrix A with columns $a_1, \ldots, a_d$, that is

$$A = \begin{pmatrix} | & & | \\ a_1 & \cdots & a_d \\ | & & | \end{pmatrix} \in \mathbb{R}^{n \times d},$$

we have

$$A^T M A = \begin{pmatrix} a_1^T M a_1 & a_1^T M a_2 & \cdots & a_1^T M a_d \\ a_2^T M a_1 & a_2^T M a_2 & \cdots & a_2^T M a_d \\ \vdots & \vdots & \cdots & \vdots \\ a_d^T M a_1 & a_d^T M a_2 & \cdots & a_d^T M a_d \end{pmatrix}.$$

So M is psd if and only if for any $A \in \mathbb{R}^{n \times d}$, we have $diag(A^T M A) = \left(a_1^T M a_1, \ldots, a_d^T M a_d\right)^T \succeq 0$, where $\succeq$ is elementwise inequality, and 0 is a $d \times 1$ column vector of 0's .

1. Use the definition of a psd matrix and the spectral theorem to show that all eigenvalues of a positive semidefinite matrix M are non-negative. [Hint: By Spectral theorem, $\Sigma = Q^T M Q$ for some Q. What if you take $A = Q$ in the "exercise in matrix multiplication" described above?]

2. In this problem, we show that a psd matrix is a matrix version of a non-negative scalar, in that they both have a "square root". Show that a symmetric matrix M can be expressed as $M = BB^T$ for some matrix B, if and only if M is psd. [Hint: To show $M = BB^T$ implies M is psd, use the fact that for any vector v, $v^T v \geq 0$. To show that M psd implies $M = BB^T$ for some B, use the Spectral Theorem.]

B Positive Definite Matrices

Definition. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is **positive definite** (spd) if for any $x \in \mathbb{R}^n$ with $x \neq 0$,

$$x^T M x > 0.$$

1. Show that all eigenvalues of a symmetric positive definite matrix are positive. [Hint: You can use the same method as you used for psd matrices above.]

2. Let M be a symmetric positive definite matrix. By the spectral theorem, $M = Q \Sigma Q^T$, where Σ is a diagonal matrix of the eigenvalues of M. By the previous problem, all diagonal entries of Σ are positive. If $\Sigma = diag\,(\sigma_1, \ldots, \sigma_n)$, then $\Sigma^{-1} = diag\left(\sigma_1^{-1}, \ldots, \sigma_n^{-1}\right)$. Show that the matrix $Q \Sigma^{-1} Q^T$ is the inverse of M.

3. Since positive semidefinite matrices may have eigenvalues that are zero, we see by the previous problem that not all psd matrices are invertible. Show that if M is a psd matrix and I is the identity matrix, then $M + \lambda I$ is symmetric positive definite for any $\lambda > 0$, and give an expression for the inverse of $M + \lambda I$.

4. Let M and N be symmetric matrices, with M positive semidefinite and N positive definite. Use the definitions of psd and spd to show that $M + N$ is symmetric positive definite. Thus $M + N$ is invertible. (Hint: For any $x \neq 0$, show that $x^T (M + N) x > 0$. Also note that $x^T (M + N) x = x^T M x + x^T N x$.)