DS-GA 1003 - Homework 3

Eric Niblock

February 24th, 2021

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.

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 $g \in \partial f(\boldsymbol{x})$.**

   Since we have that $g \in \partial f_k(\boldsymbol{x})$, then by the definition of the subgradient we have,

   $$f_k(\boldsymbol{z}) \geq f_k(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}) \tag{1}$$

   Furthermore, since we have that $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, it is obvious that,

   $$f(\boldsymbol{z}) \geq f_k(\boldsymbol{z}) \geq f_k(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}) = f(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}) \tag{2}$$

   $$f(\boldsymbol{z}) \geq f(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}) \tag{3}$$

Which shows that g is a subgradient of f at $\boldsymbol{x}$ and therefore that $g \in \partial f(\boldsymbol{x})$.

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

The gradient of $J(\boldsymbol{w})$ is well defined everywhere except when $y\boldsymbol{w}^T = 1$. In this case, we can take either gradient since is will still result in a lower bounding function. So,

$$\boldsymbol{g} = \begin{cases} -y\boldsymbol{x} & \text{for } y\boldsymbol{w}^T\boldsymbol{x} < 1 \\ 0 & \text{for } y\boldsymbol{w}^T\boldsymbol{x} \geq 1 \end{cases} \tag{4}$$

Is a valid subgradient of the hinge loss objective.

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
      $w_{t+1} = (1 - \eta_t\lambda)w_t$

---

3. 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

[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.

gradient where it is defined.

The gradient is not defined when $y_i \boldsymbol{w}^T \boldsymbol{x}_i = 1$. Otherwise, the gradient is given by,

$$\nabla_{\boldsymbol{w}} J_i(\boldsymbol{w}) = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i > 1. \end{cases} \tag{5}$$

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

$$\boldsymbol{g}_{\boldsymbol{w}} \quad = \quad \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \textbf{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \textbf{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1. \end{cases}$$

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.)

We have that,

$$J_i(\boldsymbol{w}) = f_\lambda(\boldsymbol{w}) + f_J(\boldsymbol{w}) = \frac{\lambda}{2} \|\boldsymbol{w}\|^2 + \max\left\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\right\} \tag{6}$$

Where we are using $f_J(\boldsymbol{w})$ to represent the hinge loss objective employed previously. Then we have that $\partial J_i(\boldsymbol{w}) = \partial f_\lambda(\boldsymbol{w}) + \partial f_J(\boldsymbol{w})$ since $f_\lambda(\boldsymbol{w})$ and $f_J(\boldsymbol{w})$ are both convex. Furthermore, $f_\lambda(\boldsymbol{w})$ is differentiable everywhere, and,

$$\nabla_{\boldsymbol{w}} f_\lambda(\boldsymbol{w}) = \lambda \boldsymbol{w} \tag{7}$$

We then simply add the subgradient achieved for $f_J(\boldsymbol{w})$ and find that,

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

Is a valid subgradient of $J_i(w)$.

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}`.

5. **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.**

   The following function was created to produce sparse bag-of-words representations,

```
def count_vect(list_of_words):
    vect = collections.Counter(list_of_words)
    return(vect)
```

6. **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.**

   We defined this following function specifically to create the training and testing data,

```
def train_test_split(data):
    all_x = []
    all_y = []
    for d in data:
        words = d[:-1]
        yi = d[-1]
        xi = count_vect(words)
```

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

```
        all_x.append(xi)
        all_y.append(yi)
    X_train = all_x[:1500]
    y_train = all_y[:1500]
    X_test = all_x[1500:]
    y_test = all_y[1500:]
    return(X_train, y_train, X_test, y_test)
```

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.

7. **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.**

   Below is our first implementation of the Pegasos algorithm,

```
def pegasos(X, y, lambd, epochs = 1):

    w = {}
    t = 1
    for e in range(epochs):
        for i in range(len(X)):
            t += 1
            n = 1/(t*lambd)
            xi = X[i]
            yi = y[i]
            if yi*dotProduct(w, xi) < 1:
                increment(w, -1*n*lambd, w)
                increment(w, n*yi, xi)
```

```
            else:
                increment(w, -1*n*lambd, w)
    return(w)
```

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 $s_{t+1} = (1 - \eta_t\lambda)\, s_t$.

8. **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. [4]

First we show that the new update method is equivalent to the original method. We begin by using the fact that $w_{t+1} = s_{t+1}W_{t+1}$,

$$W_{t+1} = W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j \tag{8}$$

$$\frac{w_{t+1}}{s_{t+1}} = \frac{w_t}{s_t} + \frac{1}{s_{t+1}}\eta_t y_j x_j \tag{9}$$

Now we multiply the equation by s_{t+1},

$$w_{t+1} = w_t\frac{s_{t+1}}{s_t} + \eta_t y_j x_j \tag{10}$$

Then, substituting in $s_{t+1} = (1 - \eta_t\lambda)\, s_t$ yields,

[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.

$$w_{t+1} = w_t \frac{(1 - \eta_t \lambda)\, s_t}{s_t} + \eta_t y_j x_j \tag{11}$$

$$w_{t+1} = w_t \,(1 - \eta_t \lambda) + \eta_t y_j x_j \tag{12}$$

Which is the desired equivalency.

The following is the implementation of the Pegasos algorithm with the (s, W) representation,

```
def pegasos_scale(X, y, lambd, epochs = 1):

    W = {}
    t = 1
    s = 1
    for e in range(epochs):
        for i in range(len(X)):
            t += 1
            n = 1/(t*lambd)
            xi = X[i]
            yi = y[i]
            if s*yi*dotProduct(W, xi) < 1:
                s = (1 - n*lambd)*s
                increment(W, n*yi/s, xi)
            else:
                s = (1 - n*lambd)*s

    w = {}
    increment(w, s, W)
    return(s,W,w)
```

9. 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.

The following code runs both implementations of the Pegasos algorithm and times each,

```
from timeit import default_timer as timer

start = timer()
weights = pegasos(X_train, y_train, 0.3,2)
```

```
middle = timer()
s, W, w = pegasos_scale(X_train, y_train, 0.3,2)
end = timer()

print('Pegasos Time, 2 Epochs: ', middle-start)
print('Pegasos Scaled Time, 2 Epochs: ', end-middle)
```

The following result is the elapsed time for each algorithm, in seconds,

```
Pegasos Time, 2 Epochs:  18.116645700000007
Pegasos Scaled Time, 2 Epochs:  0.3862086999999974
```

It was also confirmed that both algorithms gave the same result by examining the resulting weight vectors.

10. 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 sign$(\boldsymbol{w}^T\boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

The following function reports the classification error given a weight vector and a test set,

```
def classification_error(w, X, y):

    error = 0
    for i in range(len(X)):
        xi = X[i]
        yi = y[i]
        result = dotProduct(w, xi)
        if result > 0:
            pred = 1
        else:
            pred = -1
        if pred != yi:
            error += 1

    return(error/len(X))
```

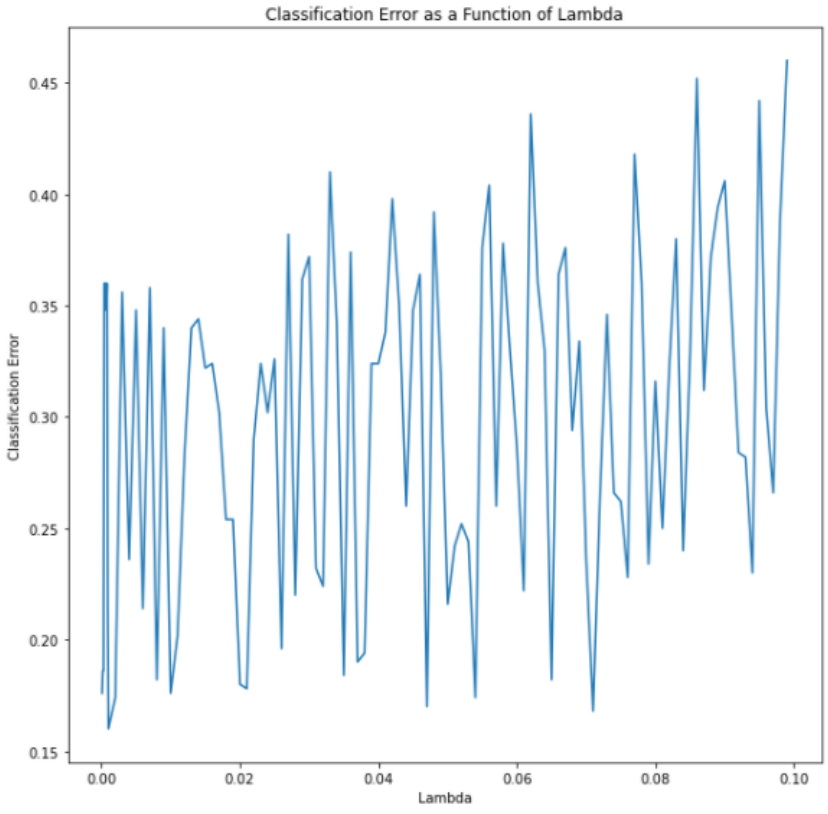
11. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. 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.)

The following code was used to determine classification error as a function of λ,

```
errs = []
ls = np.arange(0.00001,0.1,0.0001)
for lambd in ls:
    s,W,w = pegasos_scale(X_train, y_train, lambd,10)
    err = classification_error(w, X_test, y_test)
    errs.append(err)
```

We found that the best selection of parameter led to $\lambda = 0.001$ with a classification error of 0.16. Observe the following plot of classification error as a function of λ,



Though λ oscillates quite frequently, it is clear that the classification error generally increases as λ increases. Furthermore, using a λ below $\lambda = 0.01$ also leads to an increase in classification error.

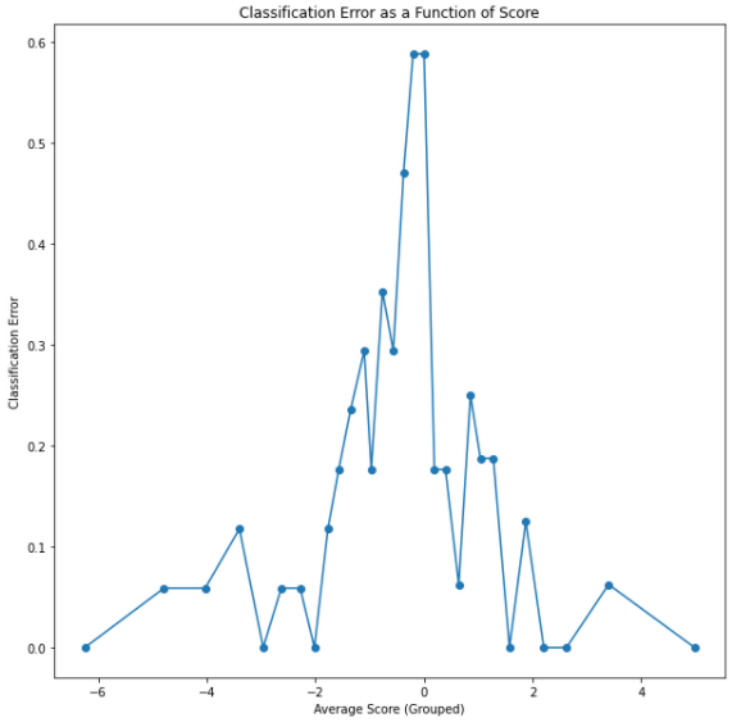
Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute.

12. **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?**

The following function was created to divide the test set scores into equal groups, and then report the classification error on each group individually. The output contains the average score per group, alongside of the groups respective classification error,

```
def classification_error_grouped(w, X, y,group_size):

    results = []
    for i in range(len(X)):
        xi = X[i]
        yi = y[i]
        result = dotProduct(w, xi)
        results.append(result)

    product_sorted, y_sorted = zip(*sorted(zip(results, y)))
    grouped = np.array_split(product_sorted, group_size)
    group_y = np.array_split(y_sorted, group_size)

    all_errs = []
    for e in range(len(grouped)):
        error = 0
        group = grouped[e]
        gy = group_y[e]
        for r in range(len(group)):
            if group[r] > 0:
                pred = 1
            else:
                pred = -1
            if pred != gy[r]:
                error += 1
        all_errs.append(error/len(group))

    return([np.mean(g) for g in grouped],all_errs)
```

The following plot shows the result of using 20 groups with an equal number of test scores. The group's score was averaged, and paired with the respective classification error,



It is easy to see that scores which lie close to the decision boundary (score=0) have a much higher classification error than those which lie further away from the decision boundary. Therefore, the larger the magnitude of the score, the stronger our confidence in the prediction.

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.

13. **(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.)

Two functions were created to assist with this problem. The first, given here, identifies the incorrectly classified examples within a test set,

```
def misclassified(w, X, y):

    wrong = []
    for i in range(len(X)):
        xi = X[i]
        yi = y[i]
        result = dotProduct(w, xi)
        if result > 0:
            pred = 1
        else:
            pred = -1
        if pred != yi:
            wrong.append((xi,yi))

    return(wrong)
```

The next function takes an individual input example, and produces the contribution to the score from each feature,

```
def feature_scores(xi, w):

    features = xi.keys()
    feature_weights = {}

    for f in features:
        temp = {}
        temp[f] = xi[f]
        feature_weights[f] = dotProduct(w, temp)
    return(feature_weights)
```

By use of these two functions, we produced two tables, one regarding a false positive example and one regarding a false negative example. For each example, we reported the 15 most significant features by contribution and sign, given here,

Example One - False Positive

Feature	Score Contribution
movie	-2.26318245450301
only	-2.2581827878141336
this	-1.7148856742883747
on	-1.5048996733551048
bad	-1.4232384507699407
was	-1.2999133391107194
no	-1.291580561295905
by	-0.9532697820145279
all	-0.9449370041997152
to	-0.9332711152589865
who	-0.9166055596293576
have	-0.8932737817478805
script	-0.8299446703553053
could	-0.6349576694886967
be	-0.5399640023998389
despite	0.41997200186653993
first	0.4499700019998648
from	0.5132991133924362
well	0.6199586694220349
best	0.6899540030664582
great	0.8099460035997554
the	0.8416105592960059
book	0.8749416705552917
with	0.8982734484367614
that	0.8999400039997347
one	0.9299380041330552
own	1.1165922271848485
he	3.0547963469101926
is	3.2614492367175236
and	5.572961802546477

Example Two - False Negative

Feature	Score Contribution
bad	-1.4232384507699407
be	-0.944937004199718
only	-0.9032731151256534
no	-0.7749483367775429
this	-0.7349510032664464
to	-0.6066262249183412
who	-0.4583027798146788
do	-0.4266382241183896
script	-0.41497233517765264
all	-0.40497300179987794
?	-0.386640890607291
should	-0.3833077794813656
by	-0.346643557096192
there	-0.3466435570961917
movie	-0.3233117792147157
true	0.14165722285180912
almost	0.14165722285180912
performance	0.14165722285180932
from	0.14665688954069606
force	0.1933204453036458
it's	0.2049863342443822
he	0.23498433437770713
the	0.2499833344443582
also	0.2633157789480685
jedi	0.2766482234517686
bit	0.31331244583694234
very	0.47996800213318896
one	0.6199586694220368
is	1.2015865608959297
and	1.4665688954069676

There are likely many different reasons why our model incorrectly predicted these two examples. Firstly, notice that the false positive is heavily influenced by '*and*'. Here is a classic example of where we should apply domain knowledge, recognizing that conjunctions and other less descriptive (and more functionally oriented) words should be removed or down-weighted within our classifier (other examples include '*is*', '*the*', and '*by*'). Some of these words could be filtered out during pre-processing.

Another failure of our classifier is the inability to recognize negations of speech. By using only singular words as features, we are unable to capture expressions such as '*not good*' and '*not bad*'. Examining example one, which is falsely labeled positive, it's likely that the feature '*great*' is proceeded by a negation. We find a similar problem in example two, with '*bad*' and '*not bad*'. We could employ n-grams, which allows for features that are longer than simply one word. This would allow us to capture not only the use of negative language, but also more linguistic meaning within the text itself.

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:

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

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 ψ. 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.

[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 $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

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 $\lambda > 0$.

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

We find the gradient as follows,

$$\begin{aligned} \nabla J(\boldsymbol{w}) &= \nabla(X\boldsymbol{w} - y)^T(X\boldsymbol{w} - y) + \lambda \boldsymbol{w}^T\boldsymbol{w} \\ &= 2X^T \cdot (X\boldsymbol{w} - y) + 2\lambda\boldsymbol{w} \end{aligned} \tag{13}$$

Then setting the gradient equal to zero allows us to achieve the minimizer. So,

$$2X^TX\boldsymbol{w} - 2X^Ty + 2\lambda\boldsymbol{w} = 0 \tag{14}$$

$$X^TX\boldsymbol{w} + \lambda\boldsymbol{w} = X^Ty \tag{15}$$

Which is the desired result. Then, if we assume that $X^TX + \lambda I$ is invertible, we would have the following,

$$(X^TX + \lambda I)\boldsymbol{w} = X^Ty \tag{16}$$

$$\boldsymbol{w} = (X^TX + \lambda I)^{-1}X^Ty \tag{17}$$

Now, we must show that $X^TX + \lambda I$ is invertible. We know that postive definite matrices are always invertible. So, if we show that $X^TX + \lambda I$ is positive definite, then this suffices to show that it is invertible. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is positive definite if for any $\boldsymbol{x} \in \mathbb{R}^n$ with $\boldsymbol{x} \neq \mathbf{0}$,

$$\boldsymbol{x}^T M \boldsymbol{x} > 0.$$

Therefore, we show that,

$$\boldsymbol{x}^T(X^TX + \lambda I)\boldsymbol{x} > 0 \tag{18}$$
$$\boldsymbol{x}^T X^T X \boldsymbol{x} + \boldsymbol{x}^T \lambda I \boldsymbol{x} > 0 \tag{19}$$
$$(X\boldsymbol{x})^T X \boldsymbol{x} + \lambda \boldsymbol{x}^T \boldsymbol{x} > 0 \tag{20}$$
$$||X\boldsymbol{x}|| + \lambda ||\boldsymbol{x}|| > 0 \tag{21}$$

Where the last expression is trivially true since $||X\boldsymbol{x}||$ is non-negative (magnitudes cannot be negative) and $\lambda||\boldsymbol{x}||$ is strictly positive since $\lambda > 0$ and $\boldsymbol{x} \neq \mathbf{0}$. Therefore $X^TX + \lambda I$ is postive definite and invertible.

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

Beginning with the provided expression we have,

$$X^TXw + \lambda Iw = X^Ty \tag{22}$$
$$\lambda Iw = X^Ty - X^TXw \tag{23}$$
$$\boldsymbol{w} = \frac{1}{\lambda}(X^Ty - X^TXw) \tag{24}$$

Then we simply factor out X^T and distribute λ to achieve,

$$\boldsymbol{w} = X^T\left(\frac{y}{\lambda} - \frac{Xw}{\lambda}\right) \tag{25}$$

So we have that,

$$\boldsymbol{\alpha} = \frac{y}{\lambda} - \frac{Xw}{\lambda} \tag{26}$$

Where we could even replace w with our previous solution such that $\boldsymbol{\alpha} = f(X, y, \lambda)$.

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

Given that,

$$X = \begin{bmatrix} -\boldsymbol{x}_1- \\ \vdots \\ -\boldsymbol{x}_n- \end{bmatrix} \implies X^T = \begin{bmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{bmatrix} \tag{27}$$

We have that,

$$\boldsymbol{w} = X^T\boldsymbol{\alpha} = \begin{bmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} = \alpha_1\boldsymbol{x}_1 + ... + \alpha_n\boldsymbol{x}_n \tag{28}$$

To be "in the span of the data" would imply that $\boldsymbol{w}$ is a linear combination of the data vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$. This is exactly what $\boldsymbol{w} = \alpha_1\boldsymbol{x}_1 + ... + \alpha_n\boldsymbol{x}_n$ means.

17. Show that $\boldsymbol{\alpha} = (\lambda I + XX^T)^{-1}y$. Note that XX^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 $\boldsymbol{\alpha}$.)

Given that,

$$\boldsymbol{\alpha} = \frac{y}{\lambda} - \frac{Xw}{\lambda} \tag{29}$$

We can replace w with $X^T\boldsymbol{\alpha}$, yielding,

$$\boldsymbol{\alpha} = \frac{y}{\lambda} - \frac{XX^T\boldsymbol{\alpha}}{\lambda} \tag{30}$$

$$\lambda\boldsymbol{\alpha} + XX^T\boldsymbol{\alpha} = y \tag{31}$$

$$(\lambda I + XX^T)\boldsymbol{\alpha} = y \tag{32}$$

$$\boldsymbol{\alpha} = (\lambda I + XX^T)^{-1}y \tag{33}$$

This is the desired expression for $\boldsymbol{\alpha}$.

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

We have that,

$$\begin{aligned} X\boldsymbol{w} &= XX^T\boldsymbol{\alpha} \\ &= XX^T(\lambda I + XX^T)^{-1}y \end{aligned} \tag{34}$$

Which is the expression for α in terms of the kernel matrix XX^T.

19. 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.)

The prediction of a new point $\boldsymbol{x}$ would simply be given by,

$$\hat{f}(\boldsymbol{x}) = k_{\boldsymbol{x}}^T\alpha^* \tag{35}$$

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`.

20. **Write functions that compute the RBF kernel $k_{\mathbf{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\mathbf{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\mathbf{linear}}(x, x') = \langle x, x' \rangle$, 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 $M_{ij} = k(W_{i.}, X_{j.})$. 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`.**

The following code is the completion of the Gaussian and polynomial kernel functions,

```
def RBF_kernel(X1,X2,sigma):
    """
    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
    """

    d = scipy.spatial.distance.cdist(X1, X2, 'sqeuclidean')
    return(np.exp((-1*d)/(2*(sigma**2))))


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
    """

    return((offset + np.dot(X1,np.transpose(X2)))**degree)
```

21. 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.

The following code returns the linear kernel matrix for $\mathcal{D}_X$,

```
X1 = np.array([[-4,-1,0,2]]).T
X2 = X1

linear_kernel(X1, X2)
```

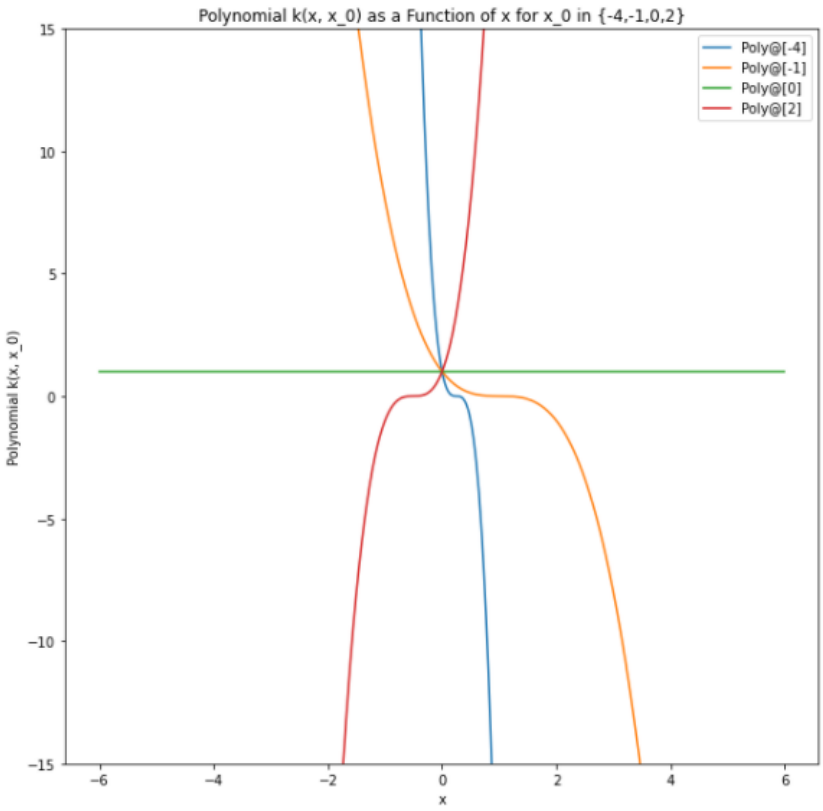
And we receive the following result,

```
array([[16,  4,  0, -8],
       [ 4,  1,  0, -2],
       [ 0,  0,  0,  0],
       [-8, -2,  0,  4]])
```

22. 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 $x \mapsto k_{\mathbf{linear}}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$ has been provided for the linear kernel.

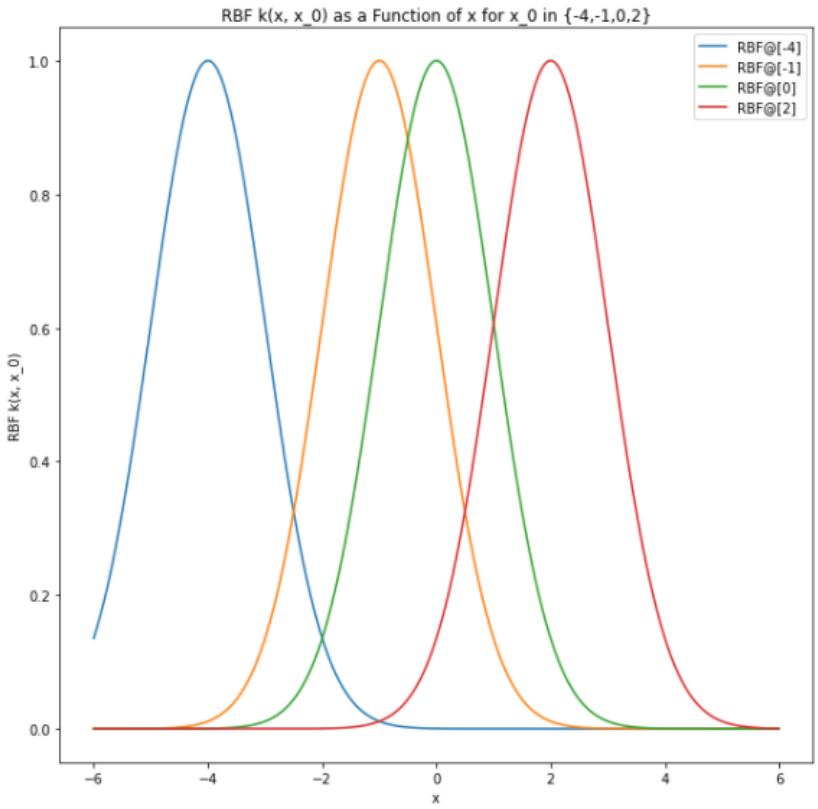
(a) Plot the set of functions $x \mapsto k_{\mathbf{poly(1,3)}}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

The following is the plot of the polynomial kernel concerning $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$ with degree 3 and offset 1,



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

The following is the plot of the RBF kernel concerning $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$ with a sigma of 1,



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

23. 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.

First, we provide the completed prediction function for our class,

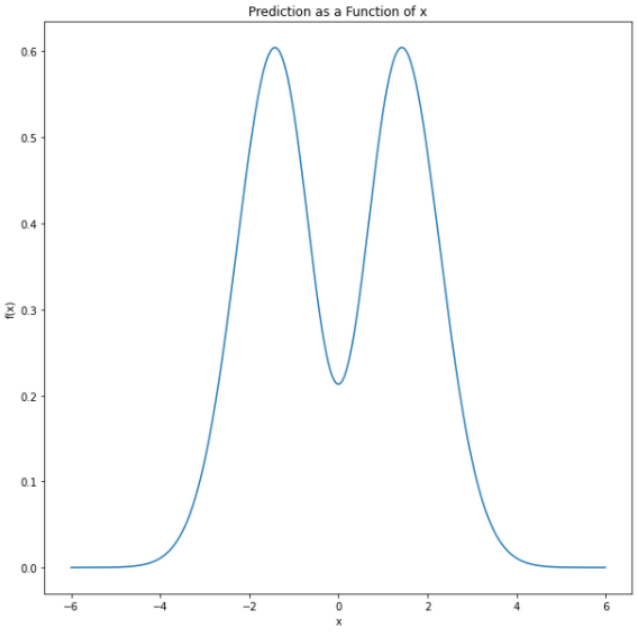
```
def predict(self, X):
        """
        Evaluates the kernel machine on the points given by the rows of X
        Args:
            X - an nxd matrix with inputs x_1,...,x_n in the rows
```

```
        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)
        """

        K0 = self.kernel(self.training_points,X)
        return(K0.T @ self.weights)
```

Next, we create an instance of the class given our data and weights, and then create our prediction function using the code below,

```
k = functools.partial(RBF_kernel, sigma=1)
ins = Kernel_Machine(k, np.array([[-1],[0],[1]]), np.array([[1],[-1],[1]]))

plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)

preds = ins.predict(xpts)
```

Finally, we receive the following plot of the prediction function, $f(x)$,



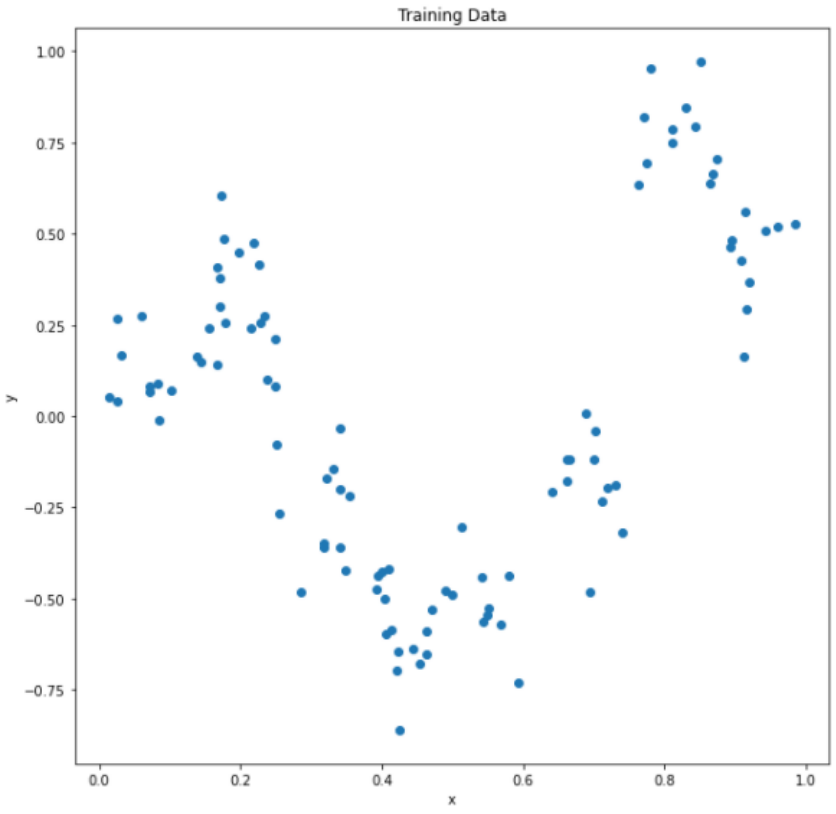
Note: For this last problem, and for other problems below, it may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`**, you can write**

`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.

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.

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

    As noted, below is the plot of the training data, which is clearly not linear,



25. **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**

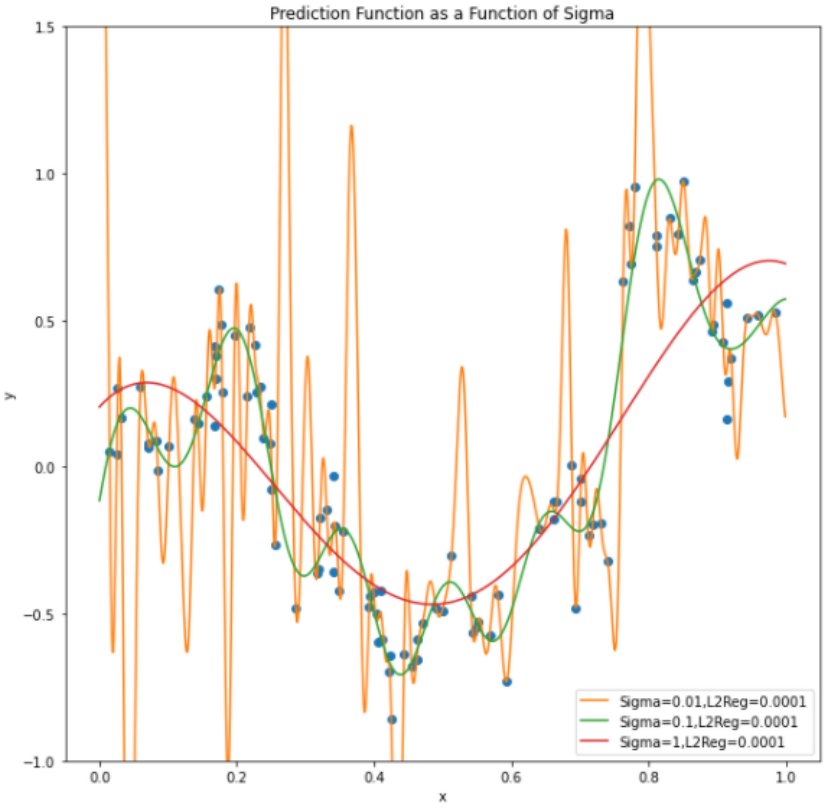
$K \in \mathbb{R}^{n \times n}$ is the kernel matrix of the training data: $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the training point $\boldsymbol{x}_i$. Complete 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.

Below is the desired function regarding kernel ridge regression,

```
def train_kernel_ridge_regression(X, y, kernel, l2reg):

    K = kernel(X,X)

    inv = np.linalg.inv(l2reg*np.identity(K.shape[0]) + K)
    alpha = inv@y

    return Kernel_Machine(kernel, X, alpha)
```

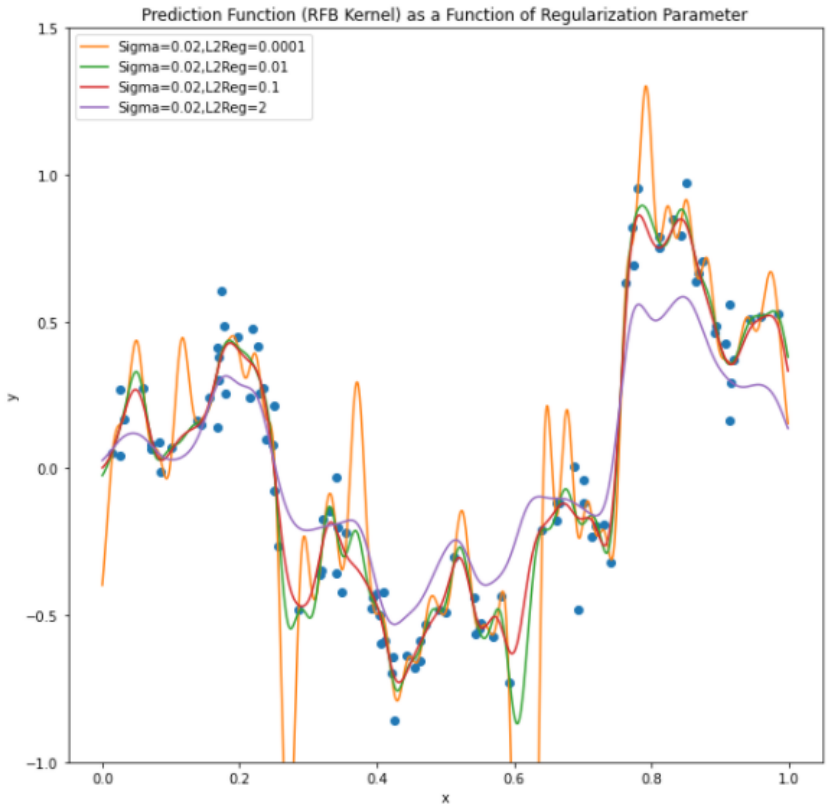
26. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed regularization parameter of 0.0001 for 3 different values of sigma: 0.01, 0.1, and 1.0. What values of sigma do you think would be more likely to over fit, and which less?

Below is the plot fitting the training data using the RBF kernel at three different values of σ and $\lambda = 0.0001$. It is clear that smaller values of σ are more likely to overfit, as the Gaussian curves become smaller, hence placing more emphasis on each individual data point, rather than the surrounding regions. Larger values of σ generate smoother distributions, with less oscillations.



27. 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** $\lambda \to \infty$**?**

Below is our code for the RFB kernel with four different values of λ and $\sigma = 0.02$. If we call our fit $g(x)$, then it is clear that as $\lambda \to \infty$, $g(x) \to 0$. For example, if we preform a fit using $\lambda = 10000$, we find that the resulting $g(x)$ is essentially constant at 0. Using such a large λ places an overbearing emphasis on making the resulting weights small, so they all tend towards zero.



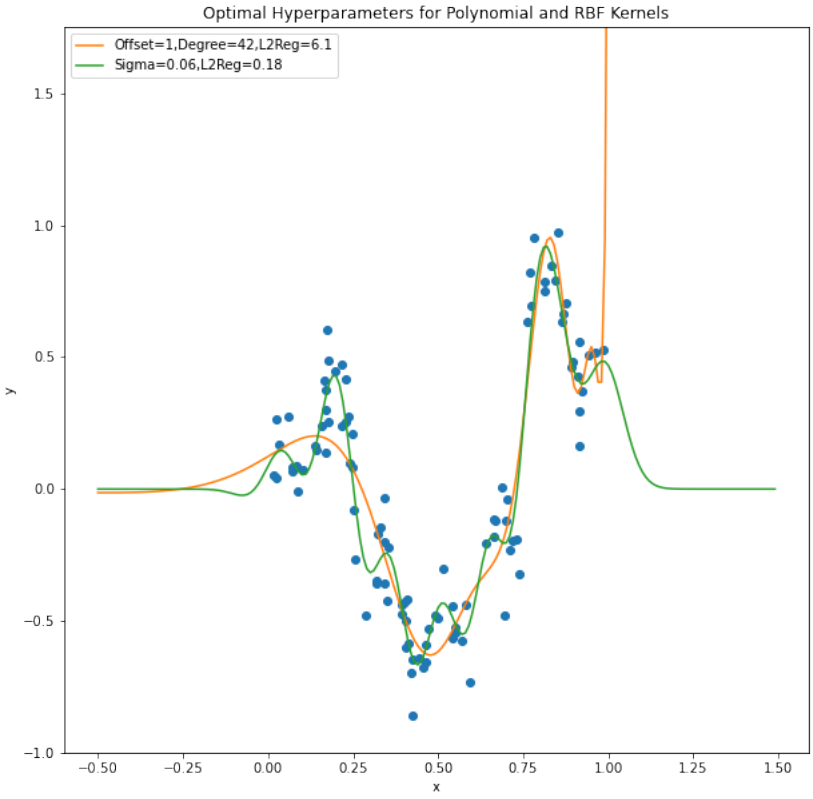
28. Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error for each setting. Include in your table the best settings for each kernel type, as well as 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.

Below are our results from extensively training the hyperparameters using the test score as our metric. We were able to obtain very small test scores. The best settings for each kernel are given in boldface within the table below,

Kernel	λ	σ	offset	d	Test Score	Train Score
RBF	0.05	0.07	—	—	0.01386	0.01432
RBF	**0.06**	**0.07**	—	—	**0.01384**	**0.01452**
RBF	0.07	0.07	—	—	0.01387	0.01473
RBF	0.06	0.06	—	—	0.01451	0.01324
RBF	**0.06**	**0.07**	—	—	**0.01384**	**0.01452**
RBF	0.06	0.08	—	—	0.01514	0.01680
Polynomial	5.5	—	1.0	42	0.02216	0.02763
Polynomial	**6.1**	—	**1.0**	**42**	**0.02215**	**0.02779**
Polynomial	7.0	—	1.0	42	0.02216	0.02802
Polynomial	6.1	—	0.9	42	0.03902	0.04790
Polynomial	**6.1**	—	**1.0**	**42**	**0.02215**	**0.02779**
Polynomial	6.1	—	1.1	42	0.02465	0.04629
Polynomial	6.1	—	1.0	41	0.02228	0.02479
Polynomial	**6.1**	—	**1.0**	**42**	**0.02215**	**0.02779**
Polynomial	6.1	—	1.0	43	0.02216	0.04030
Linear	3	—	—	—	0.16451	0.20656
Linear	**4**	—	—	—	**0.16451**	**0.20654**
Linear	5	—	—	—	0.16451	0.20659

29. 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.

The following plot shows the optimal fitting prediction functions concerning the polynomial and RBF kernels. The RBF kernel appears to fit the training data more closely, though carries higher complexity than the polynomial fit. However, the testing score associated with the RBF kernel is about half of that associated with the polynomial kernel, which seems to bear much less complexity.



30. 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 decision function is a function, f^* such that,

$$f^* \in \arg\min_{\hat{f}} R(\hat{f}) = \arg\min_{\hat{f}} E[\ell(\hat{f}(x), y)] = \arg\min_{\hat{f}} E[(\hat{f}(x) - y)^2] \tag{36}$$

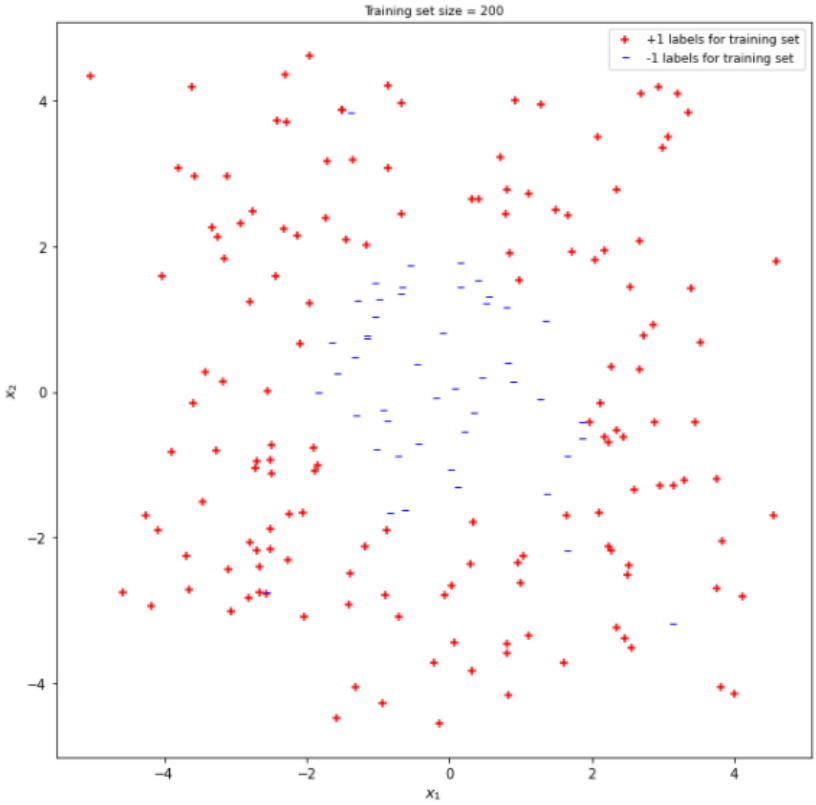
So, we proceed by calculating the expected value and then minimizing it,

$$
\begin{aligned}
E[(\hat{f}(x) - y)^2] &= E[(\hat{f}(x) - f(x) - \epsilon)^2] \\
&= E[\hat{f}(x)^2 + f(x)^2 + \epsilon^2 - 2\hat{f}(x)f(x) + 2f(x)\epsilon - 2\hat{f}(x)\epsilon] \\
&= E[\hat{f}(x)^2 + f(x)^2] + E[\epsilon^2] - 2E[\hat{f}(x)f(x)] + 2E[f(x)\epsilon - \hat{f}(x)\epsilon] \\
&= E[\hat{f}(x)^2 + f(x)^2] + Var(\epsilon) - 2E[\hat{f}(x)f(x)] + 2E[f(x)]E[\epsilon] - 2E[\hat{f}(x)]E[\epsilon] \\
&= E[\hat{f}(x)^2 + f(x)^2] - 2E[\hat{f}(x)f(x)] + Var(\epsilon)
\end{aligned}
\tag{37}
$$

Where the third line is by linearity of expectation, and the fourth line is by independence. Now, we know that the loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$ has a range of $[0, \infty)$. By inspection, We have that $\hat{f}(x) = f(x)$ yields $E[(\hat{f}(x) - y)^2] = Var(\epsilon)$. This is the lowest possible expected loss since $Var(\epsilon)$ is irreducible error, and so $\hat{f}(x) = f(x)$ is a valid Bayes decision function. So, $f^*(x) = f(x)$. The Bayes risk, which is simply the expected value of the loss given on Bayes prediction function, has already been noted to be $Var(\epsilon) = 0.01$.

31. (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?

Below we have the plot of the training data for our classification problem,



We see immediately that the data is not linearly separable (we could not draw a line that distinctly divides the two classes). On the other hand, the data is quadratically separable. We see that the decision boundary could be roughly approximated as a circle with radius 2. Therefore we could approximate $x_1^2 + x_2^2 < 2$ as a sign of being marked as the -1 class. We could achieve this combination of features using a quadratic/polynomial kernel. Similarly, we could use a RBF kernel, imagining a 3D Gaussian curve, intersecting our plane in approximately circular cross-sections (one of which could be used as a decision boundary).

32. (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.

Below we implemented the kernelized Pegasos, a version of soft SVM,

```
def train_soft_svm(X, y, kernel, lambd, epochs):
```

```
    K = kernel(X,X)
    alpha = np.zeros(K.shape[0]).reshape(-1,1)
    t=1
    for ep in range(epochs):
        for i in range(len(X)):
            ay = np.multiply(alpha,y)
            if y[i]*(1/(lambd*t))*(K[i,:]@ay) < 1:
                alpha[i]+=1
            t+=1

    return Kernel_Machine(kernel, X, np.multiply(alpha,y))
```

33. (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.

First, we wrote a function to calculate the classification error (0-1 loss) on on the testing data,

```
def classification_error(ybar,ytest):
    y1 = ybar.flatten()
    y2 = ytest.flatten()
    err = 0
    for i in range(len(y1)):
        if y1[i]*y2[i] < 0:
            err += 1
    return(err/len(y1))
```

Then, we hyper-parameter tested each kernel by looping over the following code:

```
k = functools.partial(----_kernel, -----)
f = train_soft_svm(x_train, y_train, k, lambd,15)
y_bar = f.predict(x_test)

print(classification_error(y_bar,y_test))
```

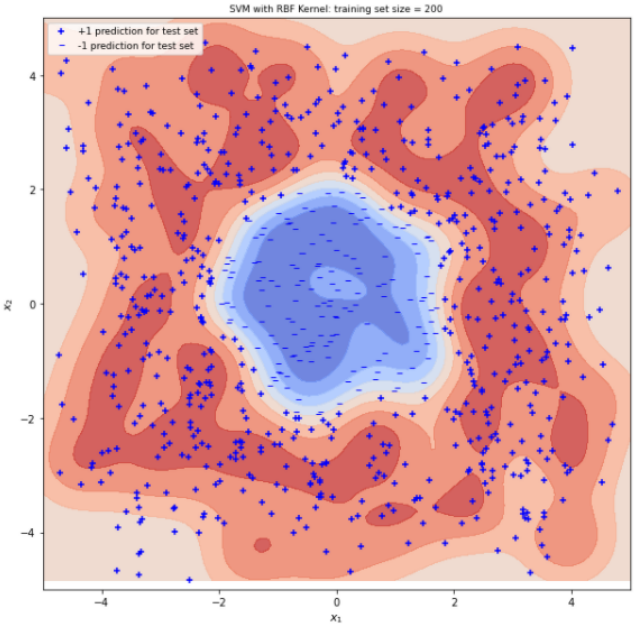
Where the dashes have been left where kernel and hyper-parameter selection are needed. The following results help to show that we selected the proper parameters,

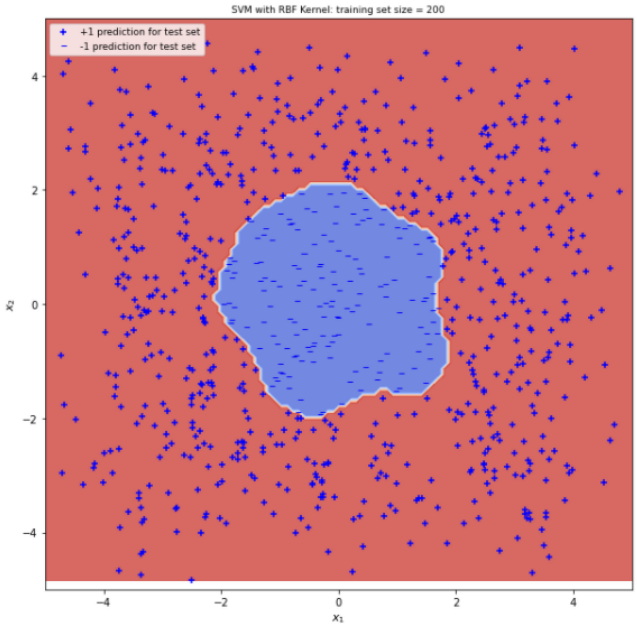
Kernel	λ	σ	**offset**	d	**Classification Error**
Linear	0.28	—	—	—	0.4938
Linear	**0.30**	—	—	—	**0.4913**
Linear	0.32	—	—	—	0.4925
RBF	0.007	0.4	—	—	0.0425
RBF	**0.007**	**0.5**	—	—	**0.0388**
RBF	0.007	0.6	—	—	0.0438
RBF	0.006	0.5	—	—	0.0400
RBF	**0.007**	**0.5**	—	—	**0.0388**
RBF	0.008	0.5	—	—	0.0413
Polynomial	0.01	—	5.9	2	0.0600
Polynomial	**0.02**	—	**5.9**	**2**	**0.0475**
Polynomial	0.03	—	5.9	2	0.1025
Polynomial	0.02	—	5.8	2	0.0600
Polynomial	**0.02**	—	**5.9**	**2**	**0.0475**
Polynomial	0.02	—	6.0	2	0.0725
Polynomial	0.02	—	5.9	3	0.0500

So, we found that the optimal settings for each kernel are: Linear: $\lambda = 0.30$; RFB: $\lambda = 0.007$, $\sigma = 0.50$; Polynomial: $\lambda = 0.20$, $\mathit{offset} = 5.9$, $\mathit{degree} = 2$.

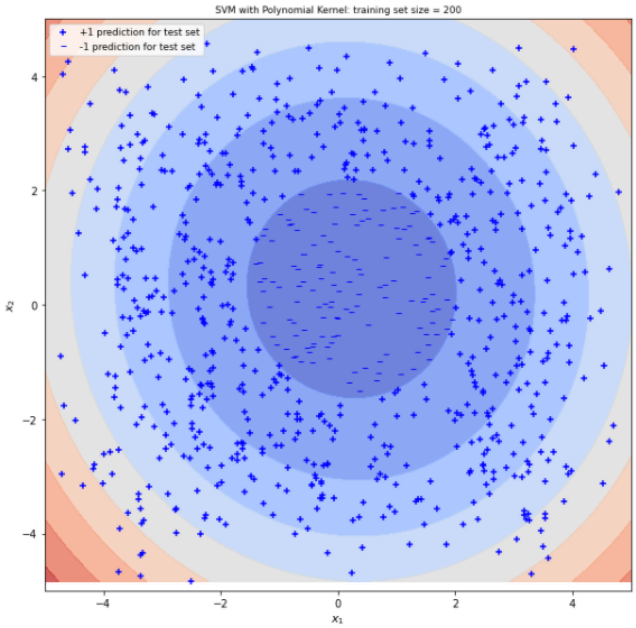
34. (Optional) Plot your best fitting prediction functions using the linear, polynomial, and the RBF kernel. The code provided may help.

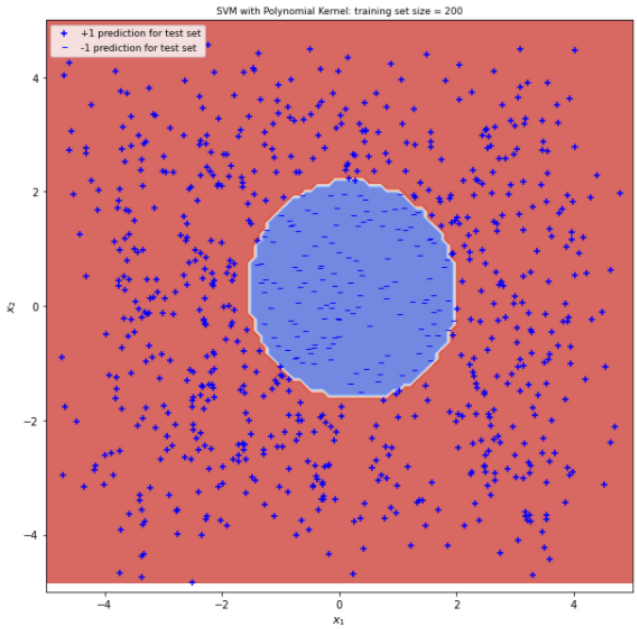
Using the provided code, we plotted the best prediction functions given the three types of kernels. Each kernel has two associated plots. The first shows the contours of the kernel, which is useful in understanding each kernel's shape. The second shows specifically the decision boundary. First, we observe the RBF kernel, which was optimal at $\sigma = 0.50$ and $\lambda = 0.007$.



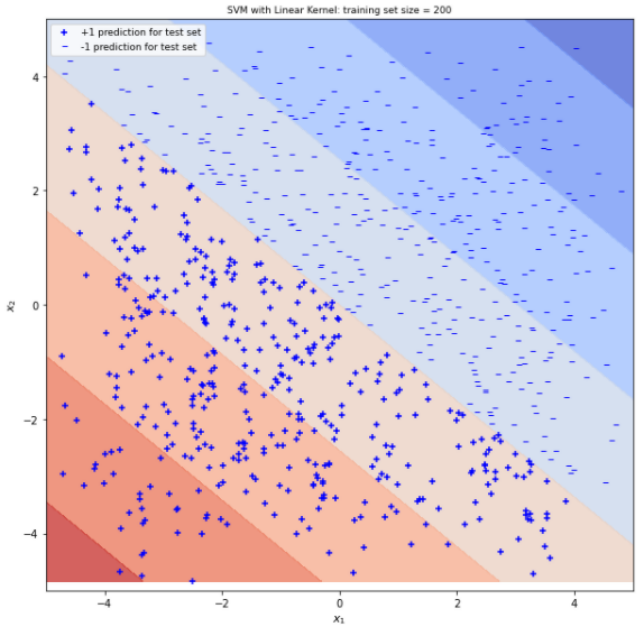


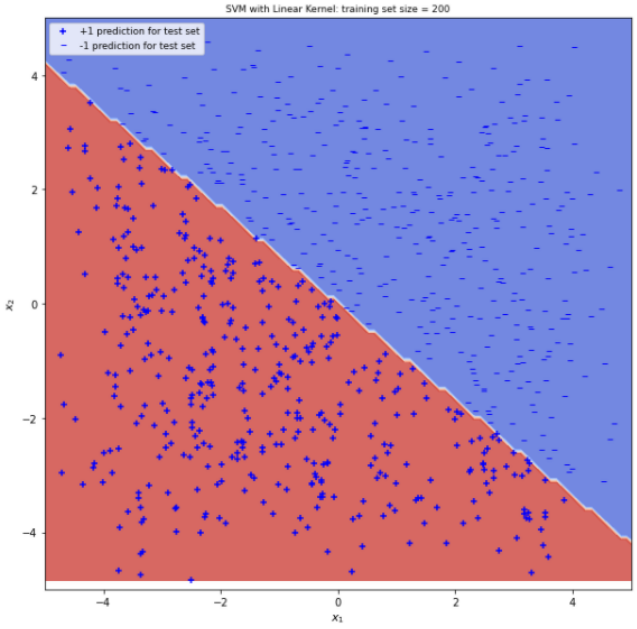
Next is the polynomial kernel, which was optimal when $d = 2$, $\mathit{offset} = 5.9$, and $\lambda = 0.02$,





And finally, the linear kernel, which is optimal when $\lambda = 0.30$


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


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