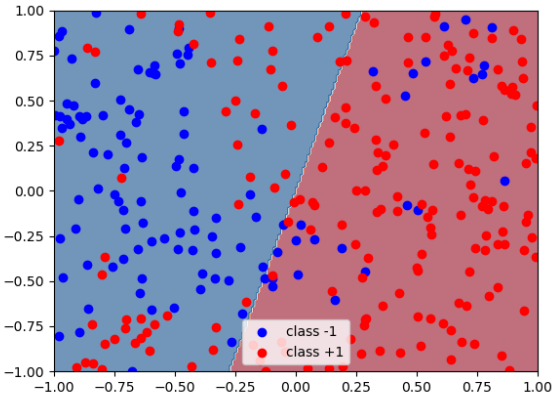
CPSC 340 Assignment 5

Important: Submission Format [5 points]

Please make sure to follow the submission instructions posted on the course website. We will deduct marks if the submission format is incorrect, or if you're not using LaTeX and your submission is *at all* difficult to read – at least these 5 points, more for egregious issues. Compared to assignment 1, your name and student number are no longer necessary (though it's not a bad idea to include them just in case, especially if you're doing the assignment with a partner).

1 Kernel Logistic Regresion [22 points]

If you run `python main.py 1` it will load a synthetic 2D data set, split it into train/validation sets, and then perform regular logistic regression and kernel logistic regression (both without an intercept term, for simplicity). You'll observe that the error values and plots generated look the same, since the kernel being used is the linear kernel (i.e., the kernel corresponding to no change of basis). Here's one of the two identical plots:



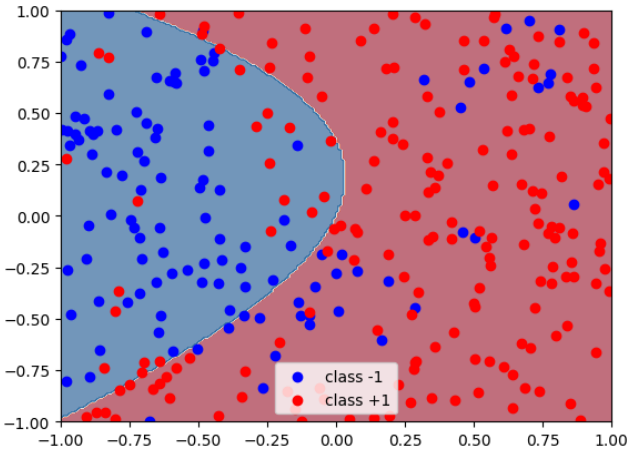
1.1 Implementing kernels [8 points]

Inside `kernels.py`, you will see classes named `PolynomialKernel` and `GaussianRBFKernel`, whose `__call__` methods are yet to be written. Implement the polynomial kernel and the RBF kernel for logistic regression. Report your training/validation errors and submit the plots from `utils.plot_classifier` for each case. You should use the kernel hyperparameters $p = 2$ and $\sigma = 0.5$ respectively, and $\lambda = 0.01$ for the regularization strength. For the Gaussian kernel, please do *not* use a $1/\sqrt{2\pi\sigma^2}$ multiplier.

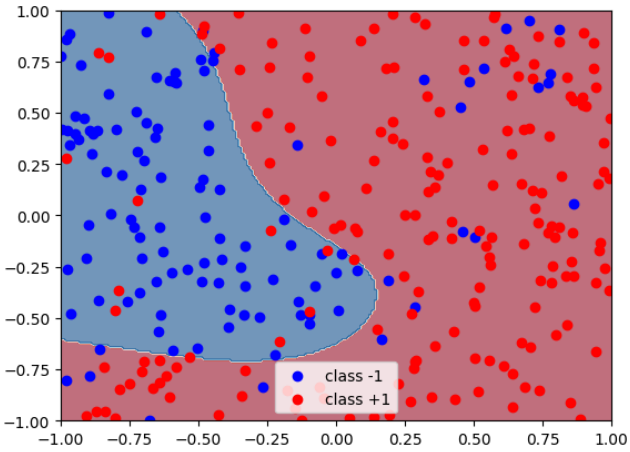
Answer:
Polynomial Kernel
Training Error: 18.3%
Validation Error: 17.0%

Gaussian RBF Kernel
Training Error: 12.7%
Validation Error: 11.0%

```python
class PolynomialKernel(Kernel):
    def __init__(self, p):
        """
        p is the degree of the polynomial
        """
        self.p = p

    def __call__(self, X1, X2):
        """
        Evaluate the polynomial kernel.
        A naive implementation will use change of basis.
        A "kernel trick" implementation bypasses change of basis.
        """

        """YOUR CODE HERE FOR Q1.1"""
        return (X1 @ X2.T + 1) ** self.p
```



```python
class GaussianRBFKernel(Kernel):
    def __init__(self, sigma):
        """
        sigma is the curve width hyperparameter.
        """
        self.sigma = sigma

    def __call__(self, X1, X2):
        """
        Evaluate Gaussian RBF basis kernel.
        """

        """YOUR CODE HERE FOR Q1.1"""
        X_distance = euclidean_dist_squared(X1, X2)
        return np.exp(-X_distance/(2*self.sigma**2))
```

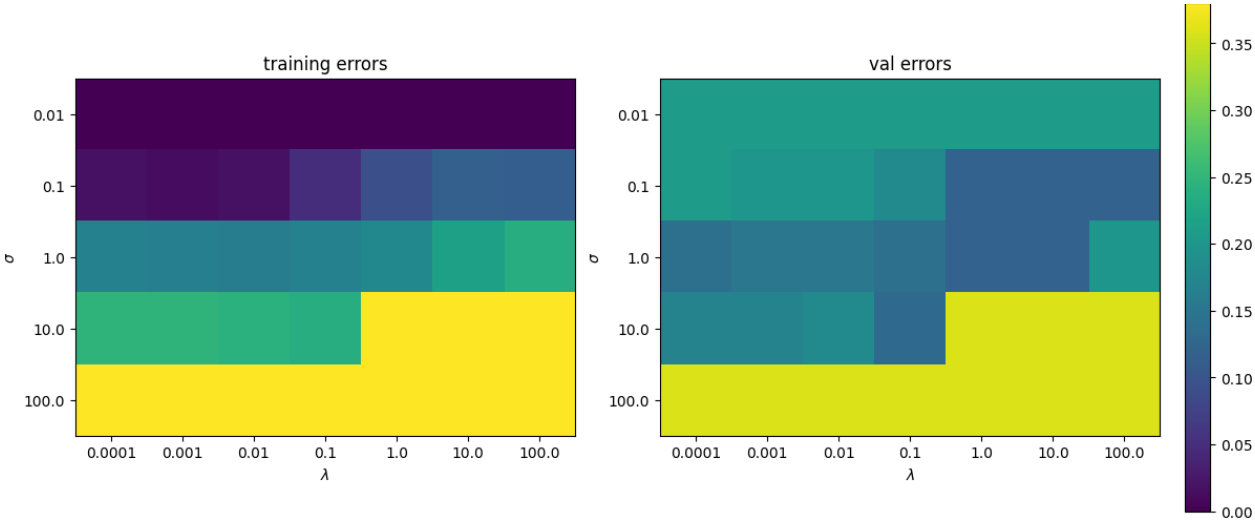


1.2 Hyperparameter search [10 points]

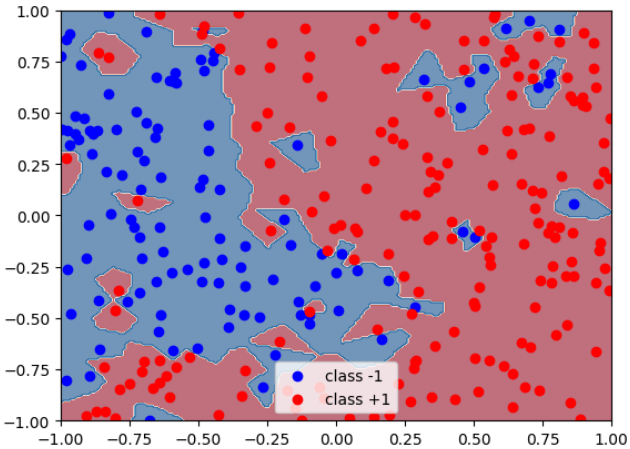
For the RBF kernel logistic regression, consider the hyperparameter values $\sigma = 10^m$ for $m = -2, -1, \ldots, 2$ and $\lambda = 10^m$ for $m = -4, -3, \ldots, 2$. The function `q1_2()` has a little bit in it already to help set up to run a grid search over the possible combination of these parameter values. You'll need to fill in the `train_errs` and `val_errs` arrays with the results on the given training and validation sets, respectively; then the code already in the function will produce a plot of the error grids. Submit this plot. Also, for each of the training and testing errors, pick the best (or one of the best, if there's a tie) hyperparameters for that error metric, and report the parameter values and the corresponding error, as well as a plot of the decision boundaries (plotting only the training set). While you're at it, submit your code. To recap, for this question you should be submitting: gridsearch plot, two decision boundary plots, the values of two hyperparameter pairs with corresponding errors, and your code.

Note: on the real job you might choose to use a tool like scikit-learn's `GridSearchCV` to implement the grid search, but here we are asking you to implement it yourself, by looping over the hyperparameter values.

Answer:

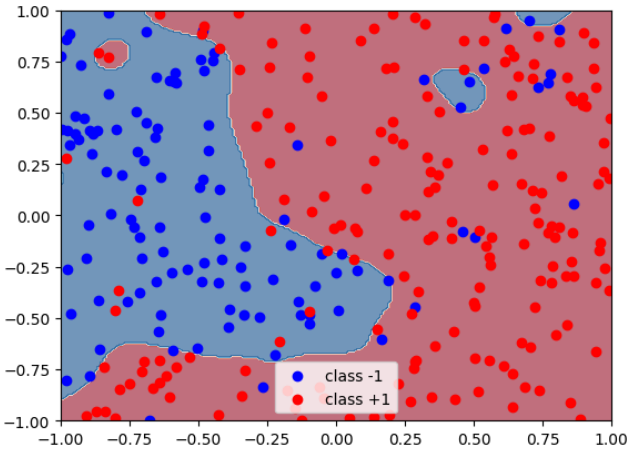


Best Training Error Decision Boundary
The best training error is 0.0% which occurs with sigma = 0.01 and lambda = 0.0001, and has a validation error of 21.0%



Best Validation Error Decision Boundary

The best validation error is 12.0% which occurs with sigma = 0.1 and lambda = 1.0, and has a training error 9.3%




```python
def q1_2():
    X_train, y_train, X_val, y_val = load_and_split("nonLinearData.pkl")
    optimizer = GradientDescentLineSearch()

    sigmas = 10.0 ** np.array([-2, -1, 0, 1, 2])
    lammys = 10.0 ** np.array([-4, -3, -2, -1, 0, 1, 2])

    # train_errs[i, j] should be the train error for sigmas[i], lammys[j]
    train_errs = np.full((len(sigmas), len(lammys)), 100.0)
    val_errs = np.full((len(sigmas), len(lammys)), 100.0)  # same for val
    min_train_err = 1
    min_train_err_i = -1
    min_train_err_j = -1

    min_val_err = 1
    min_val_err_i = -1
    min_val_err_j = -1

    for i in range(len(sigmas)):
        for j in range(len(lammys)):
            loss_fn = KernelLogisticRegressionLossL2(lammys[j])
            kernel = GaussianRBFKernel(sigmas[i])
            klr_model = KernelClassifier(loss_fn, optimizer, kernel)
            klr_model.fit(X_train, y_train)

            train_errs[i][j] = np.mean(klr_model.predict(X_train) != y_train)
            if train_errs[i][j] < min_train_err:
                min_train_err = train_errs[i][j]
                min_train_err_i = i
                min_train_err_j = j

            val_errs[i][j] = np.mean(klr_model.predict(X_val) != y_val)
            if val_errs[i][j] < min_val_err:
```

```
                min_val_err = val_errs[i][j]
                min_val_err_i = i
                min_val_err_j = j

    print(f"Min Train error: {min_train_err:.2f} at sigma value:
    ↪  {sigmas[min_train_err_i]} and lambda value: {lammys[min_train_err_j]}")
    print(f"Min Val error: {min_val_err:.2f} at sigma value: {sigmas[min_val_err_i]} and
    ↪  lambda value: {lammys[min_val_err_j]}")

    # save plot of minimum training error
    loss_fn = KernelLogisticRegressionLossL2(lammys[min_train_err_j])
    kernel = GaussianRBFKernel(sigmas[min_train_err_i])
    klr_model = KernelClassifier(loss_fn, optimizer, kernel)
    klr_model.fit(X_train, y_train)
    print(f"Training error {np.mean(klr_model.predict(X_train) != y_train):.1%}")
    print(f"Validation error {np.mean(klr_model.predict(X_val) != y_val):.1%}")
    fig = plot_classifier(klr_model, X_train, y_train)
    savefig("logRegGaussRBFMinTrainError.png", fig)

    # save plot of minimum validation error
    loss_fn = KernelLogisticRegressionLossL2(lammys[min_val_err_j])
    kernel = GaussianRBFKernel(sigmas[min_val_err_i])
    klr_model = KernelClassifier(loss_fn, optimizer, kernel)
    klr_model.fit(X_train, y_train)
    print(f"Training error {np.mean(klr_model.predict(X_train) != y_train):.1%}")
    print(f"Validation error {np.mean(klr_model.predict(X_val) != y_val):.1%}")
    fig = plot_classifier(klr_model, X_train, y_train)
    savefig("logRegGaussRBFMinValidationError.png", fig)

    # Make a picture with the two error arrays. No need to worry about details here.
```

1.3 Reflection [4 points]

Briefly discuss the best hyperparameters you found in the previous part, and their associated plots. Was the training error minimized by the values you expected, given the ways that σ and λ affect the fundamental tradeoff?

Answer: With a small σ and λ, the model is highly complex and minimally regularized. This configuration allows the model to effectively memorize the training data, leading to a training error of nearly zero. However, this comes at the cost of over fitting, as the model does not generalize well to unseen data.

On the other hand, when $\sigma = 0.01$ and $\lambda = 1$, the model becomes simpler and incorporates more regularization. This strikes a better balance between bias and variance, resulting in improved performance on the validation set. The increased λ reduces the risk of over fitting by penalizing large weights, while the small σ ensures the Gaussian RBF kernel captures local structure effectively but avoids excessive complexity.

This behavior aligns with the fundamental tradeoff: smaller σ increases model capacity, allowing finer detail to be captured, while larger λ enforces smoother and more generalized decision boundaries. The optimal hyper parameters reflect a compromise that minimizes validation error while avoiding over fitting.

2 MAP Estimation [16 points]

In class, we considered MAP estimation in a regression model where we assumed that:

- The likelihood $p(y_i \mid x_i, w)$ comes from a normal density with a mean of $w^T x_i$ and a variance of 1.
- The prior for each variable j, $p(w_j)$, is a normal distribution with a mean of zero and a variance of λ^{-1}.

Under these assumptions, we showed that this leads to the standard L2-regularized least squares objective function,

$$f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2,$$

which is the negative log likelihood (NLL) under these assumptions (ignoring an irrelevant constant). For each of the alternate assumptions below, show the corresponding loss function [each 4 points]. Simplify your answer as much as possible, including possibly dropping additive constants.

1. We use a Gaussian likelihood where each datapoint has its own variance σ_i^2, and a zero-mean Laplace prior with a variance of λ^{-1}.

$$p(y_i \mid x_i, w) = \frac{1}{\sqrt{2\sigma_i^2\pi}} \exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right), \quad p(w_j) = \frac{\lambda}{2}\exp(-\lambda|w_j|).$$

   You can use Σ as a diagonal matrix that has the values σ_i^2 along the diagonal.

   Answer:

$$w = \arg\max_w p(w_j)\Pi_{i=1}^n p(y_i \mid x_i, w)$$

$$= \arg\min_w -log(p(w_j)) - \Sigma_{i=1}^n log(p(y_i \mid x_i, w))$$

$$f(w) = -log(\frac{\lambda}{2}\exp(-\lambda|w_j|)) - \Sigma_{i=1}^n log(\frac{1}{\sqrt{2\sigma_i^2\pi}}\exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right))$$

$$= -log(\frac{\lambda}{2}) - log(\exp(-\lambda|w_j|)) - \Sigma_{i=1}^n log(\frac{1}{\sqrt{2\sigma_i^2\pi}}) + log(\exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right))$$

$$= -log(\exp(-\lambda|w_j|)) - \Sigma_{i=1}^n log(\exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right))$$

$$= \lambda|w_j| + \Sigma_{i=1}^n \frac{(w^T x_i - y_i)^2}{2\sigma_i^2}$$

$$= \lambda||w||_1 + \frac{1}{2}||\Sigma^{-1/2}(Xw - y)||_2^2$$

2. We use a Laplace likelihood with a mean of $w^T x_i$ and a variance of 8, and we use a zero-mean Gaussian prior with a variance of σ^2:

$$p(y_i \mid x_i, w) = \frac{1}{4}\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right), \quad p(w_j) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{w_j^2}{2\sigma^2}\right).$$

Answer:

$$w = \arg\max_w p(w_j)\Pi_{i=1}^n p(y_i \mid x_i, w)$$

$$= \arg\min_w -log(p(w_j)) - \Sigma_{i=1}^n log(p(y_i \mid x_i, w))$$

$$f(w) = -log(\frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{w_j^2}{2\sigma^2}\right)) - \Sigma_{i=1}^n log(\frac{1}{4}\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right))$$

$$= -log(\frac{1}{\sqrt{2\pi}\,\sigma}) - log(\exp\left(-\frac{w_j^2}{2\sigma^2}\right)) - \Sigma_{i=1}^n log(\frac{1}{4}) + log(\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right))$$

$$= -log(\exp\left(-\frac{w_j^2}{2\sigma^2}\right)) - \Sigma_{i=1}^n log(\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right))$$

$$= \frac{w_j^2}{2\sigma^2} + \Sigma_{i=1}^n \frac{1}{2}|w^T x_i - y_i|$$

$$= \frac{1}{2\sigma^2}||w||_2^2 + \frac{1}{2}||Xw - y||_1$$

3. We use a (very robust) student t likelihood with a mean of $w^T x_i$ and ν degrees of freedom, and a Gaussian prior with a mean of μ_j and a variance of λ^{-1},

$$p(y_i \mid x_i, w) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}}, \quad p(w_j) = \sqrt{\frac{\lambda}{2\pi}}\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right).$$

where Γ is the gamma function (which is always non-negative). You can use μ as a vector whose components are μ_j.

Answer:

$$w = \arg\max_w p(w_j)\Pi_{i=1}^n p(y_i \mid x_i, w)$$

$$= \arg\min_w -log(p(w_j)) - \Sigma_{i=1}^n log(p(y_i \mid x_i, w))$$

$$f(w) = -log(\sqrt{\frac{\lambda}{2\pi}}\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right)) - \Sigma_{i=1}^n log(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}})$$

$$= -log(\sqrt{\frac{\lambda}{2\pi}}) - log(\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right)) - \Sigma_{i=1}^n (log(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}) + log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}})$$

$$= \frac{\lambda}{2}(w_j - \mu_j)^2 + \Sigma_{i=1}^n \frac{\nu+1}{2} log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)$$

$$= \frac{\lambda}{2}||w - \mu||_2^2 + \frac{\nu+1}{2}\Sigma_{i=1}^n log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)$$

4. We use a Poisson-distributed likelihood (for the case where y_i represents counts), and a uniform prior for some constant κ,

$$p(y_i|w^T x_i) = \frac{\exp(y_i w^T x_i)\exp(-\exp(w^T x_i))}{y_i!}, \quad p(w_j) \propto \kappa.$$

(This prior is "improper", since $w \in \mathbb{R}^d$ but κ doesn't integrate to 1 over this domain. Nevertheless, the posterior will be a proper distribution.)

Answer:

$$w = \arg\max_w p(w_j)\Pi_{i=1}^n p(y_i \mid x_i, w)$$

$$= \arg\min_w -log(p(w_j)) - \Sigma_{i=1}^n log(p(y_i \mid x_i, w))$$

$$f(w) = -log(\kappa) - \Sigma_{i=1}^n log(\frac{\exp(y_i w^T x_i)\exp(-\exp(w^T x_i))}{y_i!})$$

$$= \Sigma_{i=1}^n \exp(w^T x_i) - (y_i w^T x_i)$$

3 Principal Component Analysis [19 points]

3.1 PCA by Hand [6 points]

Consider the following dataset, containing 5 examples with 3 features each:

x_1	x_2	x_3
0	2	0
3	-4	3
1	0	1
-1	4	-1
2	-2	2

Recall that with PCA we usually assume we centre the data before applying PCA (so it has mean zero). We're also going to use the usual form of PCA where the PCs are normalized ($\|w\| = 1$), and the direction of the first PC is the one that minimizes the orthogonal distance to all data points. We're only going to consider $k = 1$ component here.

1. What is the first principal component?

   Answer: First, we compute the mean of each feature:

   $u_1 = (0 + 3 + 1 - 1 + 2)/5 = 1$
   $u_2 = (2 - 4 + 0 + 4 - 2)/5 = 0$
   $u_3 = (0 + 3 + 1 - 1 + 2)/5 = 1$

   Then, subtract each value in the table by the corresponding mean value, to obtain the centered data:

x_1	x_2	x_3
-1	2	-1
2	-4	2
0	0	0
-2	4	-2
1	-2	1

   By observing the data and the plot, one can conclude that the first principal component before normalization is

   $$\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}.$$

   After normalizing using the denominator $\sqrt{6}$, we obtain the first principal component:

   $$w_1 = \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix}.$$

2. What is the reconstruction loss (L2 norm squared) of the point $(2.5, -3, 2.5)$? (Show your work.)

   Answer:

$$\text{First, center the new data:} \quad \tilde{X} = X - \mu = \begin{bmatrix} 2.5-1 \\ -3-0 \\ 2.5-1 \end{bmatrix}^T = \begin{bmatrix} 1.5 \\ -3 \\ 1.5 \end{bmatrix}^T.$$

   Next, find $\tilde{Z} = \tilde{X}W^T(WW^T)^{-1}$:

$$\tilde{Z} = \begin{bmatrix} 1.5 \\ -3 \\ 1.5 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \left[\tfrac{6}{6}\right]^{-1} = \frac{1.5+6+1.5}{\sqrt{6}} = \frac{9}{\sqrt{6}}.$$

   Then, plug it into reconstruction loss:

$$\|\tilde{Z}W - \tilde{X}\|_F^2 = \left\| \frac{9}{\sqrt{6}\sqrt{6}} \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} - \begin{bmatrix} 1.5 & -3 & 1.5 \end{bmatrix} \right\|_F^2$$

$$= \left\| \begin{bmatrix} \frac{9}{6} - 1.5 \\ -\frac{18}{6} + 3 \\ \frac{9}{6} - 1.5 \end{bmatrix} \right\|_F^2 = 2\left(\frac{9}{6} - 1.5\right)^2 + \left(-\frac{18}{6} + 3\right)^2) = 0.$$

3. What is the reconstruction loss (L2 norm squared) of the point $(1, -3, 2)$? (Show your work.)

   Answer:

$$\text{First, center the new data:} \quad \tilde{X} = X - \mu = \begin{bmatrix} 1-1 \\ -3-0 \\ 2-1 \end{bmatrix}^T = \begin{bmatrix} 0 \\ -3 \\ 1 \end{bmatrix}^T.$$

   Next, find $\tilde{Z} = \tilde{X}W(WW^T)^{-1}$:

$$\tilde{Z} = \begin{bmatrix} 0 \\ -3 \\ 1 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \left[\tfrac{6}{6}\right]^{-1} = \frac{6+1}{\sqrt{6}} = \frac{7}{\sqrt{6}}.$$

   Then, plug it into reconstruction loss:

$$\|\tilde{Z}W - \tilde{X}\|_F^2 = \left\| \frac{7}{\sqrt{6}\sqrt{6}} \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} - \begin{bmatrix} 0 & -3 & 1 \end{bmatrix} \right\|_F^2$$

$$= \left\| \begin{bmatrix} \frac{7}{6} \\ -\frac{14}{6} + 3 \\ \frac{7}{6} - 1 \end{bmatrix} \right\|_F^2 = \left(\frac{7}{6}\right)^2 + \left(-\frac{14}{6} + 3\right)^2 + \left(\frac{7}{6} - 1\right)^2 = \frac{11}{6}.$$

Hint: it may help (a lot) to plot the data before you start this question.

3.2 Data Visualization [7 points]

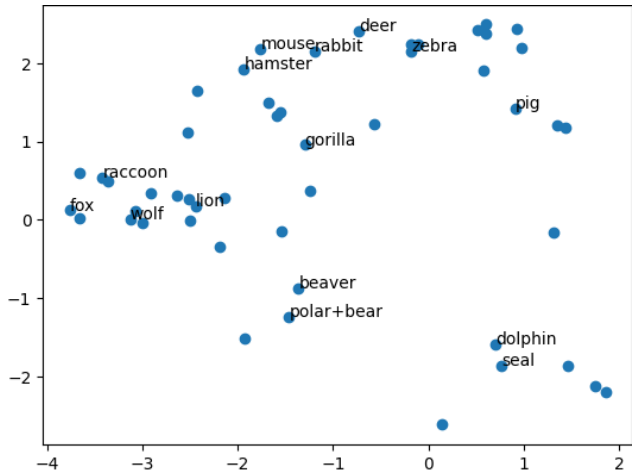
If you run `python main.py 3.2`, the program will load a dataset containing 50 examples, each representing an animal. The 85 features are traits of these animals. The script standardizes these features and gives two unsatisfying visualizations of it. First, it shows a plot of the matrix entries, which has too much information and thus gives little insight into the relationships between the animals. Next it shows a scatterplot based on two random features and displays the name of 15 randomly-chosen animals. Because of the binary features even a scatterplot matrix shows us almost nothing about the data.

In `encoders.py`, you will find a class named `PCAEncoder`, which implements the classic PCA method (orthogonal bases via SVD) for a given k, the number of principal components. Using this class, create a scatterplot that uses the latent features z_i from the PCA model with $k = 2$. Make a scatterplot of all examples using the first column of Z as the x-axis and the second column of Z as the y-axis, and use `plt.annotate()` to label the points corresponding to `random_is` in the scatterplot. (It's okay if some of the text overlaps each other; a fancier visualization would try to avoid this, of course, but hopefully you can still see most of the animals.) Do the following:

1. Hand in your modified demo and the scatterplot.

   Answer:

```
model = PCAEncoder(2)
model.fit(X_train)
W = model.W
Z = X_train @ W.T @ np.linalg.inv(W @ W.T)
print(Z)
fig, ax = plt.subplots()
ax.scatter(Z[:, 0], Z[:, 1])
for i in random_is:
    ax.annotate(animal_names[i], xy=Z[i, :])
savefig("animals_answer.png", fig)
plt.close(fig)
trait1 = trait_names[np.argmax(np.abs(W[0,:]))]
trait2 = trait_names[np.argmax(np.abs(W[1,:]))]
print(trait1, trait2)
```



2. Which trait of the animals has the largest influence (absolute value) on the first principal component?

   Answer: paws

3. Which trait of the animals has the largest influence (absolute value) on the second principal component?

   Answer: vegetation

3.3 Data Compression [6 points]

It is important to know how much of the information in our dataset is captured by the low-dimensional PCA representation. In class we discussed the "analysis" view that PCA maximizes the variance that is explained by the PCs, and the connection between the Frobenius norm and the variance of a centred data matrix X. Use this connection to answer the following:

1. How much of the variance is explained by our two-dimensional representation from the previous question?

   Answer: Variance explained: 0.221

```
 model = PCAEncoder(2)
model.fit(X_train)
W = model.W
Z = X_train @ W.T @ np.linalg.inv(W @ W.T)
X_centered = X_train - model.mu
variance_explained = 1 - np.linalg.norm((Z@W - X_centered), ord='fro')**2 /
↪  np.linalg.norm((X_centered), ord='fro')**2
print("Variance explained: {:.3f}".format(variance_explained))
```

2. How many PCs are required to explain 50% of the variance in the data?

   Answer: At k = 7 the variance explained = 0.500

```
for k in range(1,100):
    model = PCAEncoder(k)
    model.fit(X_train)
    W = model.W
    Z = X_train @ W.T @ np.linalg.inv(W @ W.T)
    X_centered = X_train - model.mu
    variance_explained = 1 - np.linalg.norm((Z@W - X_centered), ord='fro')**2 /
    ↪  np.linalg.norm((X_centered), ord='fro')**2
    if variance_explained > 0.5:
        print("at k = {:d} the variance explained = {:.3f}".format(k,
        ↪  variance_explained))
        break
```

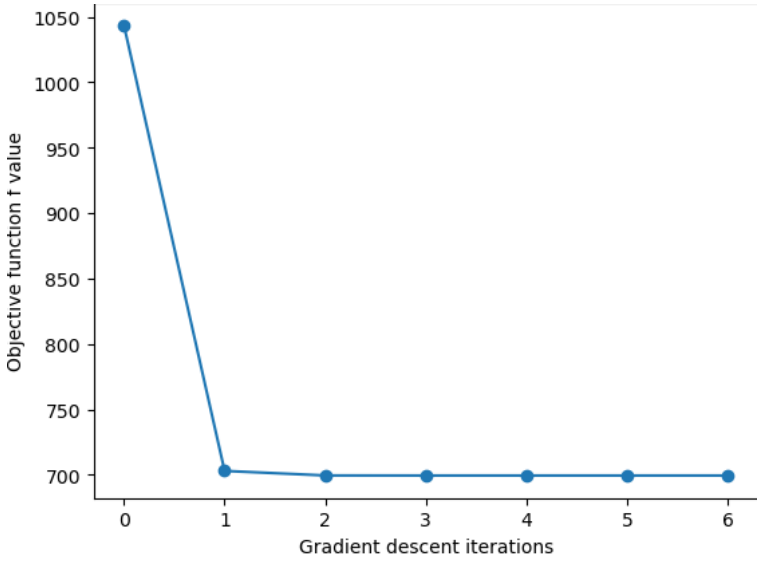
Note: you can compute the Frobenius norm of a matrix using the function `np.linalg.norm`, among other ways. Also, note that the "variance explained" formula from class assumes that X is already centred.

4 Stochastic Gradient Descent [20 points]

If you run `python main.py 4`, the program will do the following:

1. Load the dynamics learning dataset ($n = 10000, d = 5$)
2. Standardize the features
3. Perform gradient descent with line search to optimize an ordinary least squares linear regression model
4. Report the training error using `np.mean()`
5. Produce a learning curve obtained from training

The learning curve obtained from our `GradientDescentLineSearch` looks like this:



This dataset was generated from a 2D bouncing ball simulation, where the ball is initialized with some random position and random velocity. The ball is released in a box and collides with the sides of the box, while being pulled down by the Earth's gravity. The features of X are the position and the velocity of the ball at some timestep and some irrelevant noise. The label y is the y-position of the ball at the next timestep. Your task is to train an ordinary least squares model on this data using stochastic gradient descent instead of the deterministic gradient descent.

4.1 Batch Size of SGD [5 points]

In `optimizers.py`, you will find `StochasticGradient`, a *wrapper* class that encapsulates another optimizer– let's call this a base optimizer. `StochasticGradient` uses the base optimizer's `step()` method for each mini-batch to navigate the parameter space. The constructor for `StochasticGradient` has two arguments: `batch_size` and `learning_rate_getter`. The argument `learning_rate_getter` is an object of class `LearningRateGetter` which returns the "current" value learning rate based on the number of batch-wise gradient descent iterations. Currently. `ConstantLR` is the only class fully implemented.

Submit your code from `main.py` that instantiates a linear model optimized with `StochasticGradient` taking `GradientDescent` (not line search!) as a base optimizer. Do the following:

1. Use ordinary least squares objective function (no regularization).
2. Using `ConstantLR`, set the step size to $\alpha^t = 0.0003$.

3. Try the batch size values of `batch_size` $\in \{1, 10, 100\}$.

For each batch size value, use the provided training and validation sets to compute and report training and validation errors after 10 epochs of training. Compare these errors to the error obtained previously.

Answer:

Batch size: 1 Training error: 0.140 Validation error: 0.140

Batch size: 10 Training error: 0.140 Validation error: 0.140

Batch size: 100 Training error: 0.178 Validation error: 0.177

Previously we obtained a training and validation error of 0.14. For batch size of 1 and 10, the same training error and validation error was seen. For batch size 100, the training and validation error increased to 0.178 and 0.177 respectively. A bigger batch size results in faster computation time but the error does not converge compared to using smaller batch sizes which take longer to run.

```
def q4_1():
    X_train_orig, y_train, X_val_orig, y_val = load_trainval("dynamics.pkl")
    X_train, mu, sigma = standardize_cols(X_train_orig)
    X_val, _, _ = standardize_cols(X_val_orig, mu, sigma)

    """YOUR CODE HERE FOR Q4.1"""
    batch_sizes = [1,10,100]

    for batch_size in batch_sizes:
        loss_function = LeastSquaresLoss()
        base_optimizer = GradientDescent()
        learning_rate_getter = ConstantLR(0.0003)
        optimizer = StochasticGradient(base_optimizer, learning_rate_getter, batch_size,
        ↪  max_evals=10)
        model = LinearModel(loss_function, optimizer, check_correctness=False)
        model.fit(X_train, y_train)
        train_err = ((model.predict(X_train) - y_train) ** 2).mean()
        val_err = ((model.predict(X_val) - y_val) ** 2).mean()
        print("Batch size: {:d}\tTraining error: {:.3f}\tValidation error:
        ↪  {:.3f}".format(batch_size, train_err, val_err))
```

4.2 Learning Rates of SGD [6 points]

Implement the other unfinished `LearningRateGetter` classes, which should return the learning rate α^t based on the following specifications:

1. `ConstantLR`: $\alpha^t = c$.
2. `InverseLR`: $\alpha^t = c/t$.
3. `InverseSquaredLR`: $\alpha^t = c/t^2$.
4. `InverseSqrtLR`: $\alpha^t = c/\sqrt{t}$.

Submit your code for these three classes.

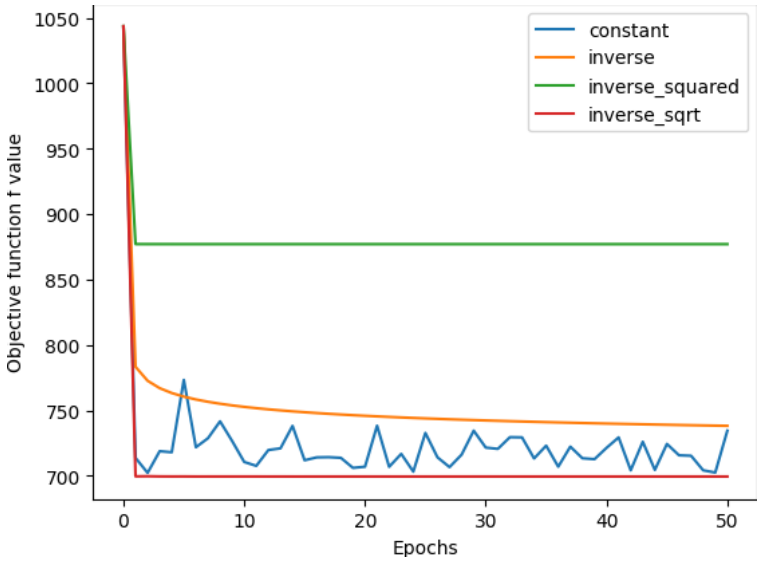
Answer:

```
class ConstantLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier


class InverseLR(LearningRateGetter):
    def get_learning_rate(self):
        """YOUR CODE HERE FOR Q4.2"""
        self.num_evals += 1
        return self.multiplier/self.num_evals


class InverseSquaredLR(LearningRateGetter):
    def get_learning_rate(self):
        """YOUR CODE HERE FOR Q4.2"""
        self.num_evals += 1
        return self.multiplier/(self.num_evals**2)


class InverseSqrtLR(LearningRateGetter):
    def get_learning_rate(self):
        """YOUR CODE HERE FOR Q4.2"""
        self.num_evals += 1
        return self.multiplier/np.sqrt(self.num_evals)
```

4.3 The Learning Curves (Again) [9 points]

Using the four learning rates, produce a plot of learning curves visualizing the behaviour of the objective function f value on the y-axis, and the number of stochastic gradient descent epochs (at least 50) on the x-axis. Use a batch size of 10. Use $c = 0.1$ for every learning rate function. Submit this plot and answer the following question. Which step size functions lead to the parameters converging towards a global minimum?

Answer:



The inverse square root step size function leads to a global minimum of around 700 which is the smallest value present in the graph.

```
    @handle("4.3")
def q4_3():
    X_train_orig, y_train, X_val_orig, y_val = load_trainval("dynamics.pkl")
    X_train, mu, sigma = standardize_cols(X_train_orig)
    X_val, _, _ = standardize_cols(X_val_orig, mu, sigma)

    """YOUR CODE HERE FOR Q4.3"""
    c = 0.1
    learning_rate_getters = [
            ConstantLR(c),
            InverseLR(c),
            InverseSquaredLR(c),
            InverseSqrtLR(c)
    ]
    plot_labels = [
        "constant",
        "inverse",
        "inverse_squared",
        "inverse_sqrt"
    ]
    plt.figure()
    for i in range(len(learning_rate_getters)):
        loss_function = LeastSquaresLoss()
        base_optimizer = GradientDescent()
        optimizer = StochasticGradient(base_optimizer, learning_rate_getters[i], 10,
        ↪  max_evals=50)
        model = LinearModel(loss_function, optimizer)
        model.fit(X_train, y_train)
        err_train = np.mean((model.predict(X_train) - y_train) ** 2)
        err_valid = np.mean((model.predict(X_val) - y_val) ** 2)
        plt.plot(model.fs, label=plot_labels[i])

    plt.legend()
    plt.xlabel("Epochs")
    plt.ylabel("Objective function f value")
    savefig("learning_curves", plt)


```

5 Very-Short Answer Questions [18 points]

Answer each of the following questions in a sentence or two.

1. Assuming we want to use the original features (no change of basis) in a linear model, what is an advantage of the "other" normal equations over the original normal equations?

   Answer: Using the "other" normal equations is an advantage when n ¡ d, i.e. there are more features than examples. This is because we would work on a n x n instead of a d x d instead, resulting in $O(n^2d+n^3)$ in training cost (compared to cubic in k^3 in normal equations) and $O(ndt)$ in testing cost.

2. In class we argued that it's possible to make a kernel version of k-means clustering. What would an advantage of kernels be in this context?

   Answer: A kernel version of k-means clustering allows for clustering in non-linear space, where clusters are separated by non-linear boundaries. This is because using kernels help capture non-linear relationships between the data and in clusters.

3. In the language of loss functions and regularization, what is the difference between MLE and MAP?

   Answer: MLE is the "likelihood", with a log-likelihood error function, and MAP is the "posterior" which takes the product of the "likelihood" and "prior". The prior introduces in the loss function of MAP a regularization term called the log-prior, which is absent in MLE.

4. What is the difference between a generative model and a discriminative model?

   Answer: A discriminative model only models a relationship between the input data and label. A generative model will learn the input data's distribution and model the relationship between the input data and the label.

5. In this course, we usually add an offset term to a linear model by transforming to a Z with an added constant feature. How, specifically, can we add an offset term to a kernel-based linear model?

   Answer: To add an offset term and ensure the intercept is not zero, we can add a constant value to the inner product result of the kernel, such as the case in this polynomial kernel expression where "1+" is added: $k(x_i, x_j) = (1 + x_i x_j^T)^p$

6. With PCA, is it possible for the loss to increase if k is increased? Briefly justify your answer.

   Answer: No. Increasing k increases the number of principal components used, meaning more variance is preserved in the resulting data. Therefore, it would not be possible for loss to increase if k increased.

7. Why doesn't it make sense to do PCA with $k > d$?

   Answer: The purpose of PCA is to reduce dimensions in data. By having more principal components compared to data features, you would be expanding the dimensions rather than reducing them, which negates the efficiency gains from using PCA rather than other models.

8. In terms of the matrices associated with PCA (X, W, Z, $\hat{X}$), where would a single "eigenface" be stored?

   Answer: An "eigenface" (eigenvector) is a column in W.

9. What is an advantage and a disadvantage of using stochastic gradient over SVD when doing PCA?

   Answer: The advantage of using stochastic gradient would require less computational resources for large datasets compared to SVD. The disadvantage of using stochastic gradient is that it may take several iterations to converge on the solution, while SVD finds an exact solution.