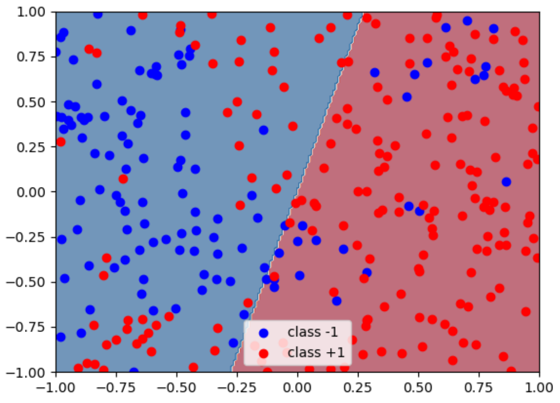
CPSC 340 Assignment 5

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 Regression [22 points]

If you run `python main.py -q 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:

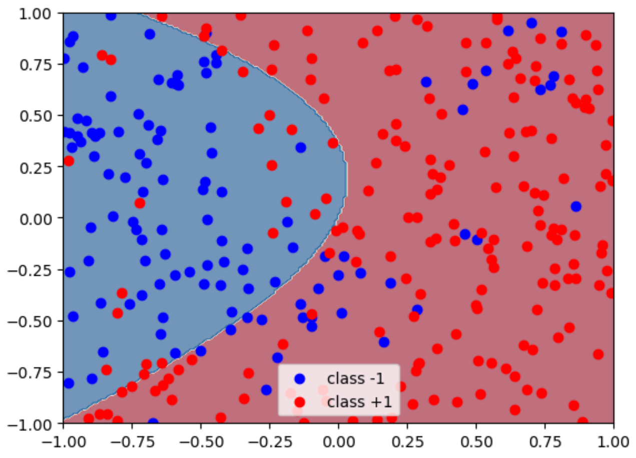


1.1 Implementing kernels [8 points]

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.

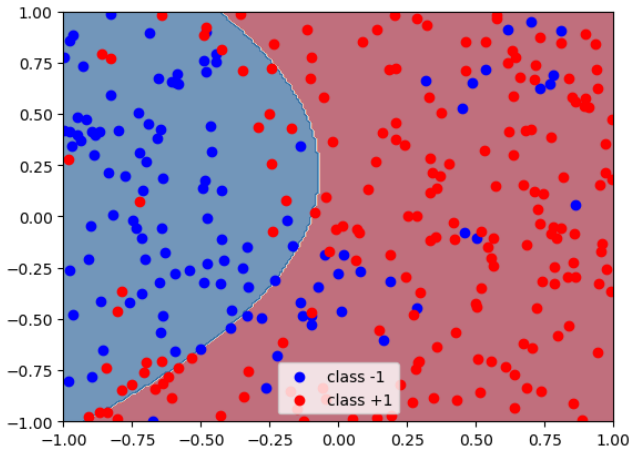
Answer: To use the kernel, we need to transform our X matrix into the Gram's Matrix. For the polynomial kernel, we can do so using $((X \times X^T) + 1)^p$, and the RBF kernel changes every value into $\exp(-\frac{1}{2\sigma^2} * \|x_i - x_j\|^2)$. After doing so, we get the following results:

For the **polynomial kernel:**



- Training error: 18.3%
- Validation error: 17%

For the **Gaussain RBF Kernel:**



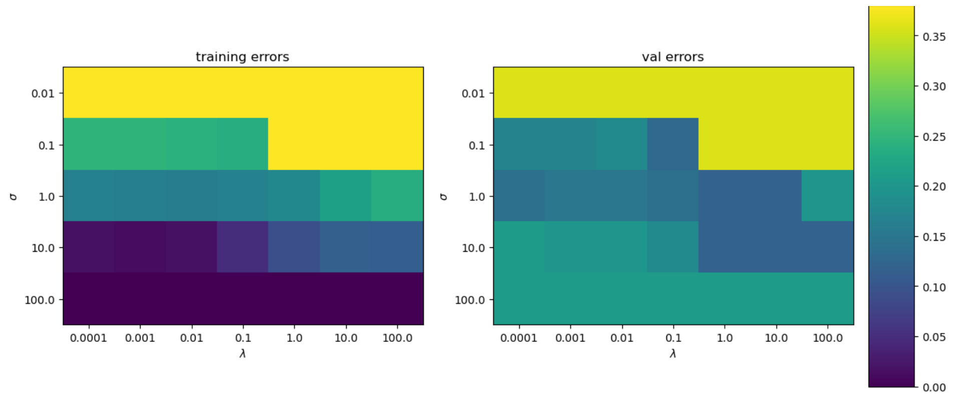
- Training error: 19.3%
- Validation error: 15%

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: two decision boundary plots, the values of two hyperparameter pairs with corresponding errors, and your code.

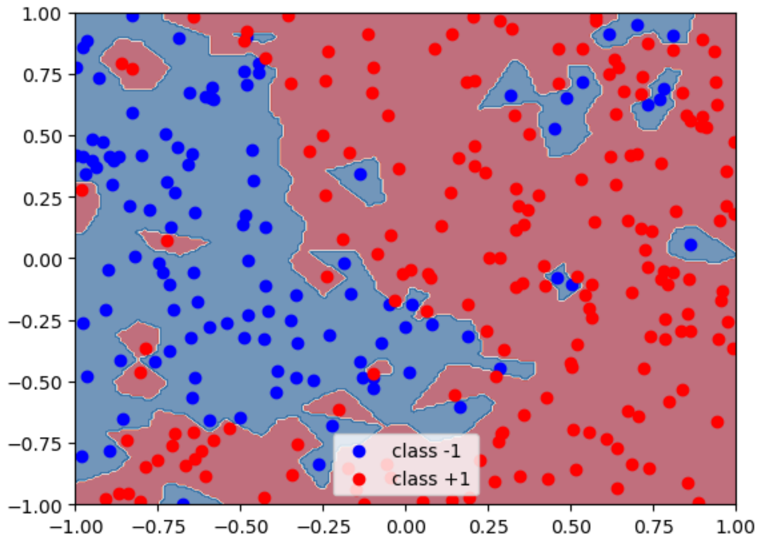
Note: on the real job you might choose to use a tool like scikit-learn's `GridSearchCV` to implement the grid search, but here we are asking you to implement it yourself, by looping over the hyperparameter values.

Answer: Here is the the grid plot:



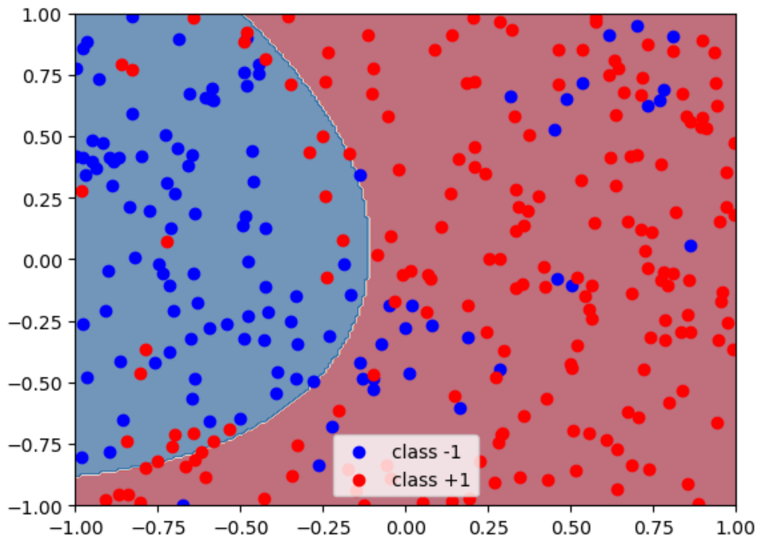
For the best values, there were quite a few ties, so I just took the hyperparameter values which are closest to (0,0) in the grid. **The best (but not-uniquely-best) hyper-parameter values:**

For training:

- $\sigma = 10$
- $\lambda = 0.001$
- Error $= 0$
- Plot:
- 

For validation:

- $\sigma = 1$
- $\lambda = 1$
- Error $= 0.12$
- Plot:

- 

Lastly, my code is below:

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

    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

    """YOUR CODE HERE FOR Q1.2"""
    for i in range(len(sigmas)):
        for j in range(len(lammys)):
            loss_fn = KernelLogisticRegressionLossL2(lammys[j])
            optimizer = GradientDescentLineSearch()
            kernel = GaussianRBFKernel(sigmas[i])
            model = KernelClassifier(loss_fn, optimizer, kernel)
            model.fit(X_train, y_train)

            train_err = np.mean(model.predict(X_train) != y_train)
            val_err = np.mean(model.predict(X_val) != y_val)

            train_errs[i,j] = train_err
            val_errs[i,j] = val_err

    # Make a picture with the two error arrays. No need to worry about details here.
    fig, axes = plt.subplots(1, 2, figsize=(12, 5), constrained_layout=True)
    norm = plt.Normalize(vmin=0, vmax=max(train_errs.max(), val_errs.max()))
    for (name, errs), ax in zip([("training", train_errs), ("val", val_errs)], axes):
        cax = ax.matshow(errs, norm=norm)

        ax.set_title(f"{name} errors")
        ax.set_ylabel(r"$\sigma$")
        ax.set_yticks(range(len(sigmas)))
        ax.set_yticklabels([str(sigma) for sigma in sigmas])
        ax.set_xlabel(r"$\lambda$")
        ax.set_xticks(range(len(lammys)))
        ax.set_xticklabels([str(lammy) for lammy in lammys])
        ax.xaxis.set_ticks_position("bottom")
    fig.colorbar(cax)
    utils.savefig("logRegRBF_grids.png", fig)


    best_train_err_index = np.unravel_index(np.argmin(train_errs, axis=None), train_errs.
    ↪                       shape)
    best_train_sigma = sigmas[best_train_err_index[0]]
    best_train_lammy = lammys[best_train_err_index[1]]
    best_train_error = train_errs[best_train_err_index]

    print(f"train_errs: {train_errs}\n")
    print(f"Best train error index: {best_train_err_index} \n")
    print(f"Best train_err sigma: {best_train_sigma}\n")
    print(f"Best train_err lambda: {best_train_lammy}\n")
    print(f"Best training error calculated: {best_train_error}")

    best_val_err_index = np.unravel_index(np.argmin(val_errs, axis=None), val_errs.shape)
    best_val_sigma = sigmas[best_val_err_index[0]]
    best_val_lammy = lammys[best_val_err_index[1]]
    best_val_error = val_errs[best_val_err_index]

    print(f"val_errs: {val_errs}\n")
    print(f"Best val error index: {best_val_err_index} \n")
    print(f"Best val_err sigma: {best_val_sigma}\n")
    print(f"Best val_err lambda: {best_val_lammy}\n")
    print(f"Best validation error calculated: {best_val_error}")

    ## plot with best params
    best_hypers = [(best_train_sigma, best_train_lammy), (best_val_sigma, best_val_lammy)
    ↪                       ]
    count = 1
    for s,l in best_hypers:
        loss_fn = KernelLogisticRegressionLossL2(l)
        optimizer = GradientDescentLineSearch()
        kernel = GaussianRBFKernel(s)
        model = KernelClassifier(loss_fn, optimizer, kernel)
        model.fit(X_train, y_train)

        fig = utils.plot_classifier(model, X_train, y_train)

        utils.savefig(f"decisionBoundary{count}", fig)

        count += 1
```

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: We can see from the grid that in **training**, larger σ produced lower error; but this is not the case for **validation** error, where error first decreases with higher σ but then increases as σ gets even higher. This also explains why the optimal σ for training error is **higher** than the optimal σ for validation error.

A similar story exists for λ. In training, lower values of λ tended to result in lower error (until σ just got very big and λ did not matter); while in validation/testing, higher values of λ usually prevailed.

Overall, we can conclude that **higher** values of σ and **lower** values of λ allow for lower training error but higher E_{approx}; while **lower** values of σ and **higher** values of λ make E_{train} higher but improves E_{approx}. More generally, lower values of σ and higher values of λ force the model to be more **generalizable** later on and not over-fit to the training data - a phenomenon we can see happening from the plots.

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:

$$\begin{aligned}
\texttt{loss\_function} &= -\sum_{i=1}^{n} \log(p(y_i \mid x_i, w)) \\
&= -\sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\sigma_i^2\pi}} \exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right)\right) \\
&= \texttt{constant} + -\sum_{i=1}^{n} \log\left(\exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right)\right) \\
&= \sum_{i=1}^{n} \left(\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right)
\end{aligned}$$

Let Σ be a diagonal matrix with σ_i^2 on the diagonal. Then, by rewriting the above expression in matrix notation, we get that:

$$\texttt{loss\_function} = \frac{1}{2}\|\Sigma^{-1}Xw - y\|^2$$

A similar process is followed for the prior:

$$\begin{aligned}
\texttt{regularizer} &= -\log(p(w)) \\
&= -\log\left(\prod_{j=1}^{d}\frac{\lambda}{2}\exp(-\lambda|w_j|)\right) \\
&= -\sum_{j=1}^{d}\left(\log\left(\frac{\lambda}{2}\exp(-\lambda|w_j|)\right)\right) \\
&= \texttt{constant} + -\sum_{j=1}^{d}(-\lambda|w_j|) \\
&= \lambda\|w\|_1
\end{aligned}$$

Therefore, the final loss function is:

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

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:

$$\begin{aligned}
\texttt{loss\_function} &= -\sum_{i=1}^{n}\log(p(y_i \mid x_i, w)) \\
&= -\sum_{i=1}^{n}\log\left(\frac{1}{4}\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right)\right) \\
&= \texttt{constant} + -\sum_{i=1}^{n} -\frac{1}{2}|w^T x_i - y_i| \\
&= \frac{1}{2}\|Xw - y\|_1
\end{aligned}$$

A similar process is followed for the prior:

$$\begin{aligned}
\texttt{regularizer} &= -\log(p(w)) \\
&= -\log\left(\prod_{j=1}^{d}\frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{w_j^2}{2\sigma^2}\right)\right) \\
&= -\sum_{j=1}^{d}\log\left(\frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{w_j^2}{2\sigma^2}\right)\right) \\
&= \texttt{constant} + -\sum_{j=1}^{d}\left(-\frac{w_j^2}{2\sigma^2}\right) \\
&= \frac{1}{2\sigma^2}\|w\|_2^2
\end{aligned}$$

Therefore, the final loss function is:

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

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:

$$\begin{aligned}
\texttt{loss\_function} &= -\sum_{i=1}^{n}\log(p(y_i \mid x_i, w)) \\
&= -\sum_{i=1}^{n}\log\left(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}}\right) \\
&= \texttt{constant} + -\sum_{i=1}^{n}\log\left(\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}}\right) \\
&= \frac{\nu+1}{2}\sum_{i=1}^{n}\log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right) \\
&= \texttt{constant} + \frac{\nu+1}{2}\sum_{i=1}^{n}\log\left(\nu + (w^T x_i - y_i)^2\right) \\
&= \frac{\nu+1}{2}\sum_{i=1}^{n}\log\left(\nu + (w^T x_i - y_i)^2\right)
\end{aligned}$$

A similar process is followed for the prior. Let μ be a $d \times 1$ vector with μ_i on entry i.

$$\begin{aligned}
\texttt{regularizer} &= -\log(p(w)) \\
&= -\log\left(\prod_{j=1}^{d}\sqrt{\frac{\lambda}{2\pi}}\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right)\right) \\
&= -\sum_{j=1}^{d}\log\left(\sqrt{\frac{\lambda}{2\pi}}\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right)\right) \\
&= \texttt{constant} + -\sum_{j=1}^{d}\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right) \\
&= \frac{\lambda}{2}\|w - \mu\|_2^2
\end{aligned}$$

Therefore, the final loss function is:

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

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

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

(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:

$$\begin{aligned}
\texttt{loss\_function} &= -\sum_{i=1}^{n} \log(p(y_i \mid w^Tx_i)) \\
&= -\sum_{i=1}^{n} \log\left(\frac{\exp(y_iw^Tx_i)\exp(-\exp(w^Tx_i))}{y_i!}\right) \\
&= \texttt{constant} + -\sum_{i=1}^{n} \log\left(\exp(y_iw^Tx_i)\exp(-\exp(w^Tx_i))\right) \\
&= -\sum_{i=1}^{n} \left((y_iw^Tx_i) - \exp(w^Tx_i)\right) \\
&= \sum_{i=1}^{n} \left(\exp(w^Tx_i) - (y_iw^Tx_i)\right)
\end{aligned}$$

A similar process is followed for the prior. Let μ be a $d \times 1$ vector with μ_i on entry i.

$$\begin{aligned}
\texttt{regularizer} &= -\log(p(w)) \\
&\propto -\log\left(\kappa^d\right) \\
&\propto \texttt{constant}
\end{aligned}$$

This means there is no regularizer, as this is **not** a function of w. Therefore, the final loss function is:

$$f(w) = \sum_{i=1}^{n} \left(\exp(w^Tx_i) - (y_iw^Tx_i)\right)$$

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: The first principal component is the vector (rounded to 5 decimal places):

$$w = \begin{bmatrix} 0.40825 \\ -0.81650 \\ 0.40825 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{6}} \\ -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix}$$

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

   Answer: First, we center the new $\hat{x} = [2.5, -3, 2.5]$ using the column means from training. The column means are $\mu = [1, 0, 1]$. Hence, our centered $\hat{x}_{cen} = [1.5, -3.1.5]$.

   Then, we calculate $z_i = \hat{x}_{cen} \cdot w_1^T = 1.5\sqrt{6} = 3.6742$ (5 d.p.)
   Finally, we calculate the reconstruction cost which is $\|z_i w - x_i\|^2 = 0$. We can see that this is the case since $\hat{x}_{cen}$ is in the span of w.

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

   Answer: Similar as above, we first center the datapoint using the training means, and we get $\tilde{x} = [0, -3, 1]$. Then, we find the inner product $\langle \tilde{x}, w \rangle = \frac{7}{\sqrt{6}}$.
   Lastly, we calculate the reconstruction error by finding $\|z_i w - x_i\|_2^2 = \frac{11}{6} = 1.83$ (2 d.p.)

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

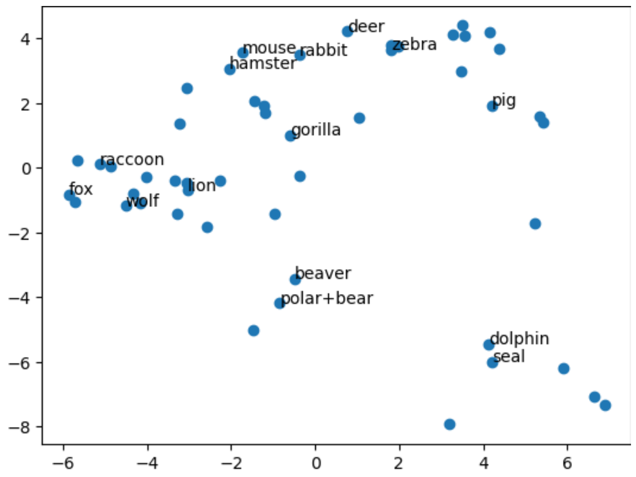
3.2 Data Visualization [7 points]

If you run `python main.py -q 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: Below is my scatter plot:



   I am unsure what "modified demo" is, but I assume that is referring to the code that I used to create this plot:

```
k = 2
pca_encoder = PCAEncoder(k)
pca_encoder.fit(X_train)
Z_train = X_train_standardized @ pca_encoder.W.T

ax.scatter(Z_train[:,0], Z_train[:,1])

for i in random_is:
    xy = Z_train[i]
    ax.annotate(animal_names[i], xy=xy)
utils.savefig("animals_PCA.png", fig)
```

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

   Answer: The trait that had the largest influence on the first principal component was "paws". What is worth noting is that this feature's value was negative, and it explains the visualization plot a bit better.

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

   Answer: The trait that had the largest influence on the first principal component was "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: This can be found by finding the ratio of $\frac{\|ZW - X\|_F^2}{\|X\|_F^2}$. I found that with 2 principal components, it explained approximately **29.172% of the variance.**

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

   Answer: To explain 50% or more of the variance in the data, we need **6 principal components**, where it explains approimately 52.6% of the total variance.

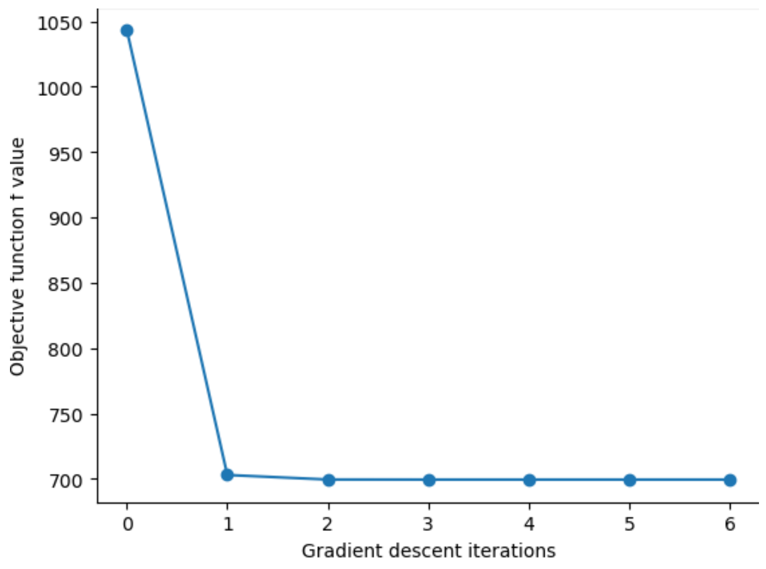
Note: you can compute the Frobenius norm of a matrix using the function `np.linalg.norm`, among other ways. Also, note that the "variance explained" formula from class assumes that X is already centred.

4 Stochastic Gradient Descent [20 points]

If you run `python main.py -q 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: The code I used was as below:

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

    # batch_size = 1
    # batch_size = 10
    batch_size = 100

    constantLRG = ConstantLR(0.0003)
    loss_fn = LeastSquaresLoss()
    optimizer = GradientDescent()
    stochastic_optimizer = StochasticGradient(optimizer, constantLRG, batch_size,
    ↪                       max_evals=10)
    model = LinearModel(loss_fn, stochastic_optimizer, check_correctness=False)
    model.fit(X_train, y_train)

    print(f"Training MSE for batch size = {batch_size}: {((model.predict(X_train) -
    ↪                       y_train) ** 2).mean():.3f}")
    print(f"Validation MSE for batch size = {batch_size}: {((model.predict(X_val) - y_val
    ↪                       ) ** 2).mean():.3f}")
```

Then, I got the following MSEs:

Batch Size	Training MSE	Validation MSE
1	0.140	0.140
10	0.140	0.140
100	0.178	0.177

Both batch size of 1 and 10 achieved the same MSE as the original, but not the batch size of 100. It is probably because the batch size = 100 method did not get enough iterations to converge to the minimum due to the large batch size.

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.

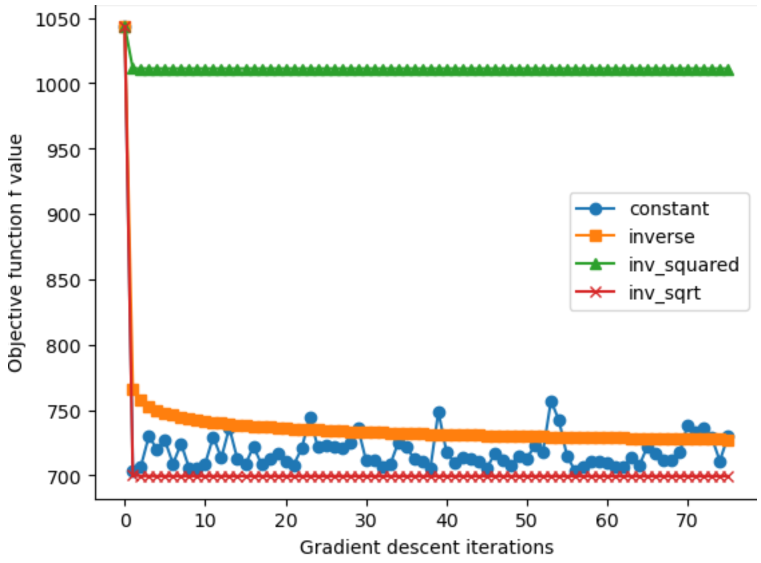
Answer: The code for these three classes are below:

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


class InverseLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier/self.num_evals


class InverseSquaredLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier/(self.num_evals ** 2)


class InverseSqrtLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier/(self.num_evals ** 0.5)
```

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: Here is my plot:



We can see that all three of `Constant`, `InverseSqrt`, and `Inverse` are converging towards what seems to be (or at least in the neighbourhood of) the global minimum. The reason why `InverseSquared` was not able to achieve this is likely because as the number of iterations increases, **the squared term amplified the effect of it**, and it causes the step size to shrink too quickly. This rapidly reduces the rate of progress towards the global minimum, and it ends up plateauing.

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: The "other" normal equations manipulates the matrix equation so that we are solving $n \times n$ matrices instead of $k \times k$. Since $k > n$, this will reduce the total computational 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: We can reduce the total computational complexity if we reduce the dimensions of the matrices we're dealing with in K-means. Also, since K-means method utilizes distances a lot, this is something that kernel methods can make a lot more efficient.

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

   Answer: The MLE uses only the likelihood of data, where MAP uses both likelihood AND the prior distribution, meaning that when we convert them to loss functions, MLE does not produce a regularizer, while MAP does. They both have the loss function component though, as that comes from the likelihood function.

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

   Answer: The main difference is what is considered "fixed". In discriminative models, we assume that the X is fixed and that we are maximizing the probability that it's label is y. In generative models, we are trying to optimize both X and y with respect to w and we consider that there to be a "likelihood of X" as opposed to discriminative models.

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 can we do that in a kernel model?

   Answer: We do so by adding a constant term in the kernel function, so that we do, for example, $k(x_i, x_j) = (1 + x_i^T x_j)^p$ for polynomial kernels.

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

   Answer: I think if we restrict what we are referring to as "PCA" to only that which is a systematic and **not**-ridiculous method (like SVD, gradient descent or other methods), then the answer is **no**. In SVD-like methods, increasing k means using more principal components, and they are all orthogonal to each other - this means that it would *at worst* offer no information (like the $(d+1)$-th orthogonal PC in d-dimentional data), but it cannot offer negative benefit because in a sense, the previous PCs are all still "there". Other optimization-based approaches would not produce orthogonal PCs but at least they would offer a new direction of subspace to project onto - also contributing positively to the loss.

   However, if we are looking at PCA in a broad, unrestricted sense, I think **it is possible**, specifically by including duplicate or completely co-linear PCs. The simplest case is going from $k = 1$ with w_1 being the first PC found from SVD; to having $k = 2$ and setting $w_2 = w_1$. I observed this when I was doing experiments by coding it, and in this case $k = 2$ reduces less variance than $k = 1$, which is a proportional measure to our loss. I cannot seem to find the mathematical explanation, but this is what I observed through experimentation.

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

   Answer: There are a few reasons. The first is that since PCA is a process of projecting onto the k-dimensional subspace, if $k > d$, the projections for the $(d+1)$-th PC would just all be zero. Plus, when $k = d$, we (vacuously) can explain all the variance of the data using the k PCs, so adding an additional one would not serve much utility.

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

   Answer: An "eigenface" would be stored in in one row of W, as one of the principal components.

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

   Answer: A disadvantage is that it becomes a lot more sensitive to initialization, and the set of vectors you get are not necessarily unique. An advantage could be that the computational complexity of stochastic gradient descent would be less than doing SVD for large matrices.