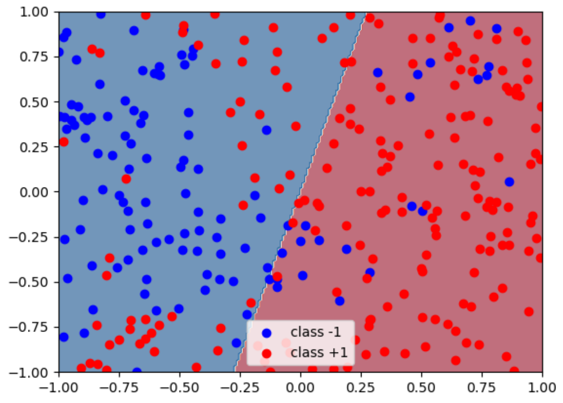
CPSC 340 Assignment 5 – due 2021-11-22

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:

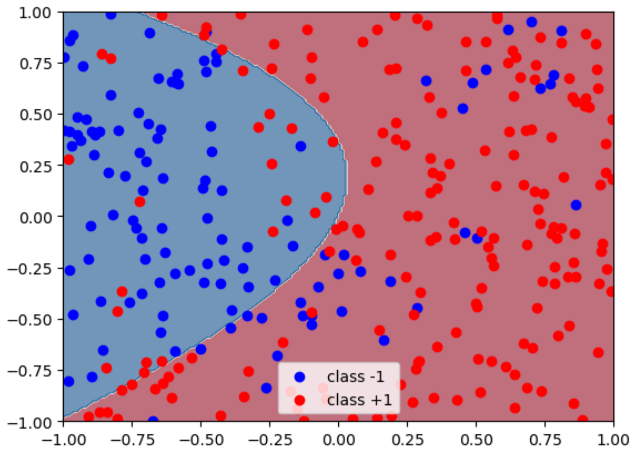


1.1 Implementing kernels [8 points]

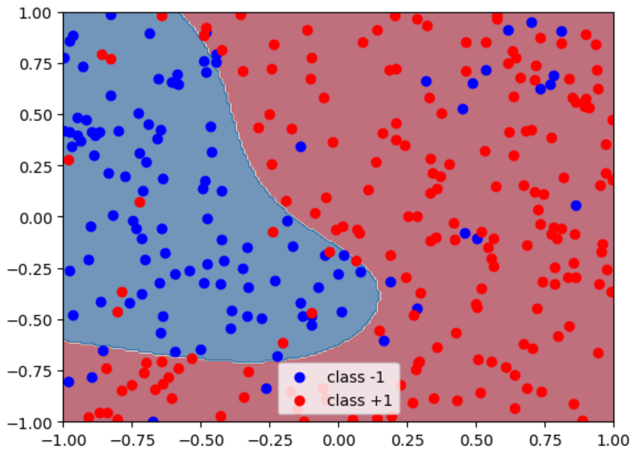
Inside `kernels.py`, you will see classes named `PolynomialKernel` and `GaussianRBFKernel`, whose `__call__` methods are yet to be written.

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:
Polynomial Kernel:
Training error: 18.3%
Validation error: 17.0%



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

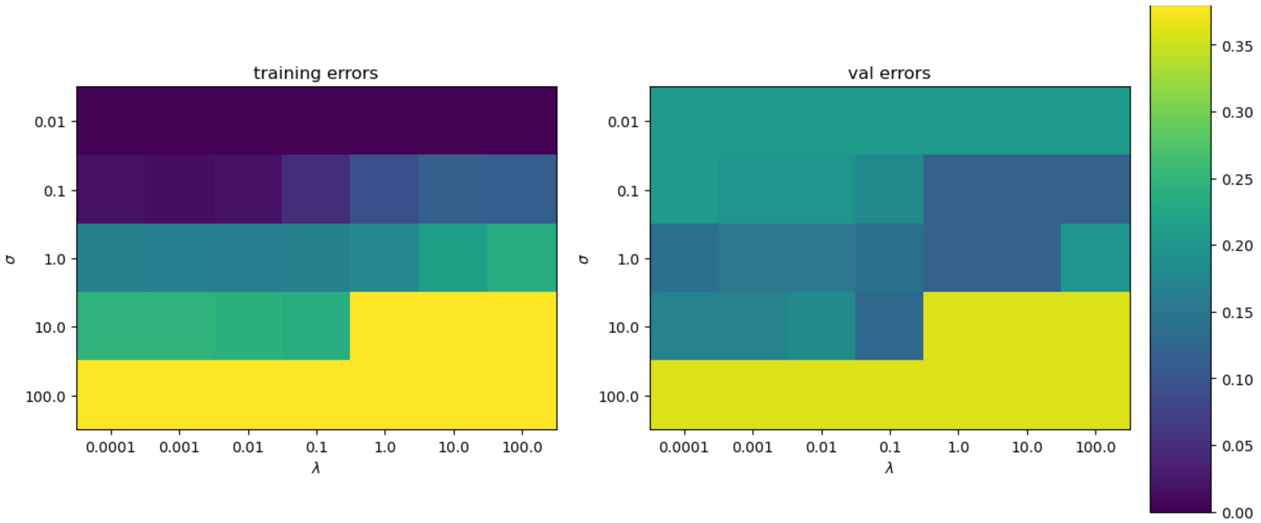


1.2 Hyperparameter search [10 points]

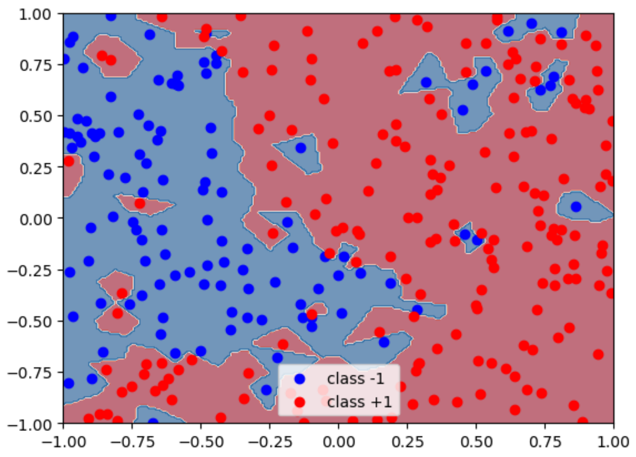
For the RBF kernel logistic regression, consider the hyperparameter values $\sigma = 10^m$ for $m = -2, -1, \dots, 2$ and $\lambda = 10^m$ for $m = -4, -3, \dots, 2$. The function `q1_2()` has a little bit in it already to help set up to run a grid search over the possible combination of these parameter 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.

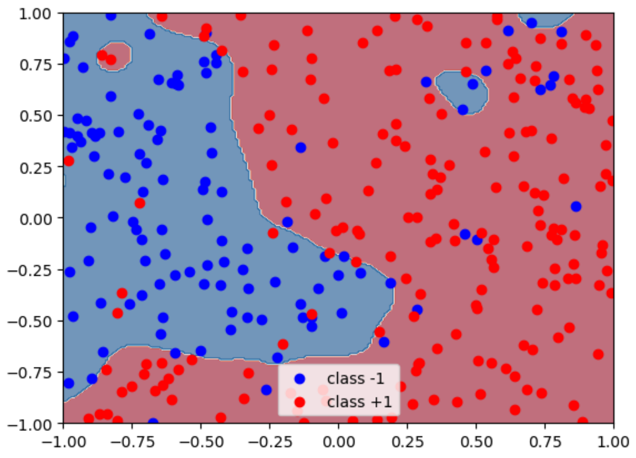
Answer:
RBF grids plot:



Parameter values from minimum training error:
Sigma - 0.01
Lambda - 0.0001
Corresponding training error: 0.0%
Corresponding validation error: 21.0%
Decision boundary plot with these values:



Parameter values from minimum validation error:
Sigma - 0.1
Lambda - 1.0
Corresponding training error: 9.3%
Corresponding validation error: 12.0%
Decision boundary plot with these values:



Code (sorry that y'all have to look at this):

```python
for i in range(len(sigmas)):
    curr_sigma = sigmas[i]
    for j in range(len(lammys)):
        loss_fn = KernelLogisticRegressionLossL2(lammys[j])
        optimizer = GradientDescentLineSearch()
        kernel = GaussianRBFKernel(curr_sigma)
        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)
        val_errs[i][j] = np.mean(klr_model.predict(X_val) != y_val)
# pulled this from
↪  https://stackoverflow.com/questions/3230067/numpy-minimum-in-row-column-format
train_i, train_j = np.unravel_index(train_errs.argmin(), train_errs.shape)
val_i, val_j = np.unravel_index(val_errs.argmin(), val_errs.shape)

train_sigma = sigmas[train_i]
train_lammy = lammys[train_j]

print(f"Training Sigma: {train_sigma}")
print(f"Training Lambda: {train_lammy}")

val_sigma = sigmas[val_i]
val_lammy = lammys[val_j]

print(f"Validation Sigma: {val_sigma}")
print(f"Validation Lambda: {val_lammy}")

# using the best param vals from the training errors
loss_fn = KernelLogisticRegressionLossL2(train_lammy)
optimizer = GradientDescentLineSearch()
kernel = GaussianRBFKernel(train_sigma)
klr_model = KernelClassifier(loss_fn, optimizer, kernel)
klr_model.fit(X_train, y_train)

```

```python
print(f"Training error from training params {np.mean(klr_model.predict(X_train) !=
↪   y_train):.1%}")
print(f"Validation error from training params {np.mean(klr_model.predict(X_val) !=
↪   y_val):.1%}")

fig = utils.plot_classifier(klr_model, X_train, y_train)
utils.savefig("logRegTrainParams.png", fig)

# using the best param vals from the validation error
loss_fn = KernelLogisticRegressionLossL2(val_lammy)
optimizer = GradientDescentLineSearch()
kernel = GaussianRBFKernel(val_sigma)
klr_model = KernelClassifier(loss_fn, optimizer, kernel)
klr_model.fit(X_train, y_train)

print(f"Training error from val params {np.mean(klr_model.predict(X_train) !=
↪   y_train):.1%}")
print(f"Validation error from val params {np.mean(klr_model.predict(X_val) !=
↪   y_val):.1%}")

fig = utils.plot_classifier(klr_model, X_train, y_train)
utils.savefig("logRegValParams.png", fig)
```

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:
Training params:
The σ and λ values that we got from the lowest training error are 0.01 and 0.0001, respectively. This gave us a training error of 0.0% and a validation error of 21.0%. The plot that resulted from this has a very jagged decision boundary that only highlights the blue points as blue, and the red points as red, which makes sense, given the training error.
In terms of the fundamental tradeoff, it makes sense that we have a small training error because our small σ refers to a smaller standard deviation for our Gaussian curves, which results in the being taller and more fitted to the data. Similarly with our small λ, this means that there is less of a regularization penalty model, meaning that it's more fitted to the data, which gives us our small training error. Since our training error is small it also makes sense for our validation error to be larger (in this question, the validation error is being used as an approximation to the approximation error).

Validation params:
The σ and λ values that we got from the lowest validation error are 0.1 and 1.0, respectively. The plot that we got from this has smoother decision boundaries and also has a few of the data points incorrectly classified, which corresponds to our higher training error.
In terms of the fundamental tradeoff, we have larger σ and λ values, which means that are model is less fitted to the training data, which gives us the higher training error (9.3%). Since our training error is higher, it also makes sense for our validation error to be lower (12.0%).

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^Tx_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}.

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

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

Answer: $\frac{1}{2}\|Xw - y\|^2 + \lambda\|w\|_1$

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

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

Answer: $f(w) = \|Xw - y\|_1 + 1/2\sigma^2 \ \|w\|^2$

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

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

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

Answer: $f(w) = \frac{v+1}{2}\sum_{i=1}^n log(1 + \frac{(w^Tx_i - y_i)^2}{\nu}) + \lambda/2\|w\|^2$

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

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.

1. What is the first principal component?

   Answer: $\begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix}$

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

   Answer:
   `https://en.wikibooks.org/wiki/Linear_Algebra/Orthogonal_Projection_Onto_a_Line`
   First, centre the point using the means for each feature from the training set.
   Let μ_1 be the mean for x_1, μ_2 be the mean for x_2, and μ_3 be the mean for x_3.
   $\mu_1 = \frac{0+3+1-1+2}{5} = 1$
   $\mu_2 = \frac{2-4+0+4-2}{5} = 0$
   $\mu_3 = \frac{0+3+1-1+2}{5} = 1$

   Now centre the point.
   $(2.5-1, -3-0, 2.5-1) = (1.5, -3, 1.5)$
   $\tilde{X} = \begin{bmatrix} 1.5 & -3.0 & 1.5 \end{bmatrix}$
   Next, we have to find our $\tilde{Z}$.
   $\tilde{Z} = \tilde{X}W^T(WW^T)^{-1}$
   $$= \begin{bmatrix} 1.5 & -3.0 & 1.5 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} \\ \frac{-2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \left(\begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} \\ \frac{-2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \right)^{-1} = \begin{bmatrix} \frac{-3}{\sqrt{6}} \end{bmatrix}$$
   Finally, we can calculate the reconstruction loss by taking the squared Frobenius norm of $\tilde{Z}W - \tilde{X}$
   $$\|\tilde{Z}W - \tilde{X}\|_F^2 = \| \begin{bmatrix} \frac{-3}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix} - \begin{bmatrix} 1.5 & -3.0 & 1.5 \end{bmatrix} \|_F^2 = \| \begin{bmatrix} \frac{-1}{2} & 1 & \frac{-1}{2} \end{bmatrix} - \begin{bmatrix} 1.5 & -3.0 & 1.5 \end{bmatrix} \|_F^2$$
   $= \| \begin{bmatrix} -2 & 4 & -2 \end{bmatrix} \|_F^2 = (-2)^2 + 4^2 + (-2)^2 = 4 + 16 + 4 = \mathbf{24}$

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

   Answer:
   Let's once again start be centering the point, using the means calculated in the previous part (wheee)
   $(1-1, -3-0, 2-1) = (0, 0, 1)$
   $\tilde{X} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ Now that we have our $\tilde{X}$, we can determine $\tilde{Z}$ using the same approach as the previous part.
   $\tilde{Z} = \tilde{X}W^T(WW^T)^{-1}$

$$= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} \\ \frac{-2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \left(\begin{bmatrix} \frac{1}{\sqrt{6}} \\ \frac{-2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \end{bmatrix} \right)^{-1} = \begin{bmatrix} \frac{1}{\sqrt{6}} \end{bmatrix}$$

Finally, we can calculate the reconstruction loss once again using the Frobenius norm, like in the previous part.

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

$$= \| \begin{bmatrix} \frac{1}{6} & \frac{-2}{6} & \frac{-5}{6} \end{bmatrix} \|_F^2 = (\tfrac{1}{6})^2 + (\tfrac{-2}{6})^2 + (\tfrac{-5}{6})^2 = \frac{1}{36} + \frac{4}{36} + \frac{25}{36} = \boldsymbol{\frac{5}{6}}$$

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

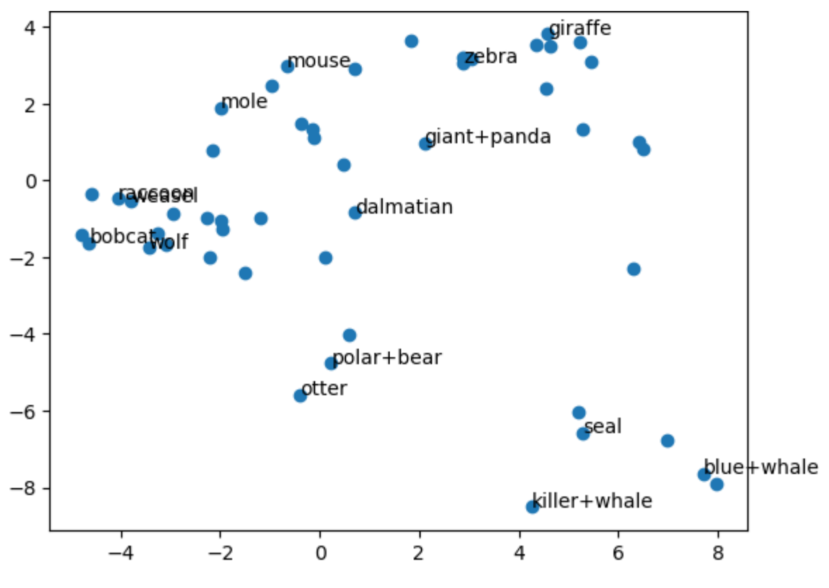
3.2 Data Visualization [7 points]

If you run `python main.py -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 `compressors.py`, you will find a class named `PCA`, 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:



```
"""YOUR CODE HERE FOR Q3"""
model = PCAEncoder(2)
model.fit(X_train)
```

```python

Z_tilde = model.encode(X_train_standardized)

# 2D visualization
np.random.seed(3164)  # make sure you keep this seed
random_is = np.random.choice(n, 15, replace=False)  # choose random examples

fig, ax = plt.subplots()
ax.scatter(Z_tilde[:, 0], Z_tilde[:, 1])
for i in random_is:
    xy = Z_tilde[i, [0, 1]]
    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: 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: 32.313%

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

   Answer: 5 or more

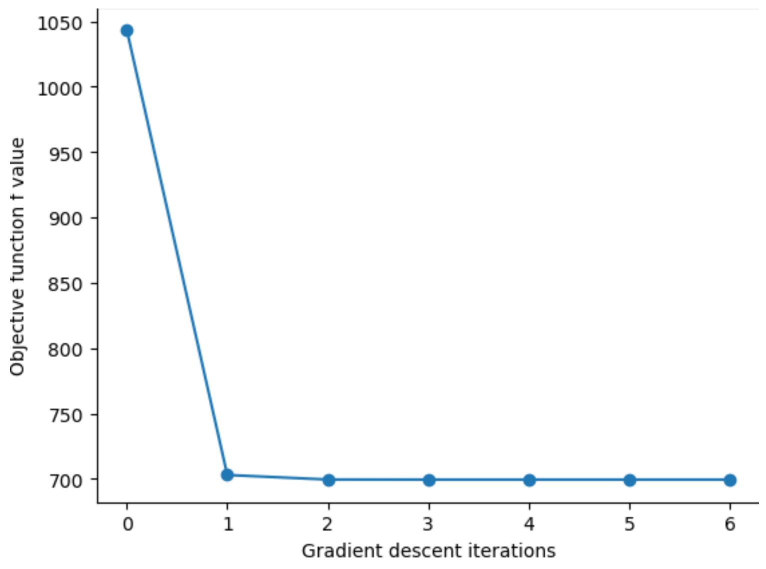
Note: you can compute the Frobenius norm of a matrix using the function `np.linalg.norm`, among other ways. Also, note that the "variance explained" formula from class assumes that X is already centred.

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` $\in \{1, 10, 100\}$.

```python
"""YOUR CODE HERE FOR Q4.1"""
    #raise NotImplementedError()
    loss_fn = LeastSquaresLoss()
    learning_rate_getter = ConstantLR
    optimizer = StochasticGradient(base_optimizer = GradientDescent(),
    ↪  learning_rate_getter = ConstantLR(multiplier = 0.0003), batch_size=100,
    ↪  max_evals=10)
    model = LinearModel(loss_fn, optimizer, check_correctness=False)
    model.fit(X_train, y_train)
    print(f"Training MSE: {((model.predict(X_train) - y_train) ** 2).mean():.3f}")
    print(f"Validation MSE: {((model.predict(X_val) - y_val) ** 2).mean():.3f}")
```

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

Both the training error and the validation error did not change when the batch size is increased from 1 to 10, but the training error and the validation error increased when batch size is increased from 10 to 100.

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.

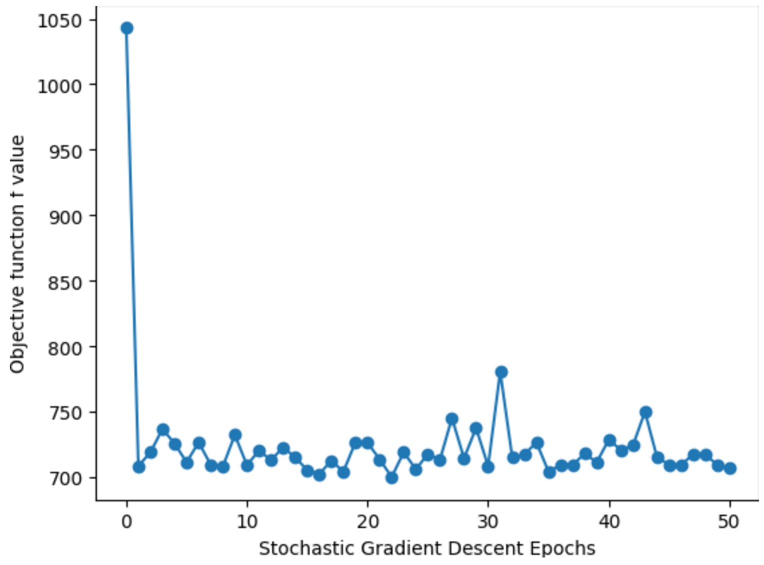
```python
class InverseLR(LearningRateGetter):
    def get_learning_rate(self):
        """YOUR CODE HERE FOR Q4.2"""
        #raise NotImplementedError()
        self.num_evals +=1
        return self.multiplier/self.num_evals


class InverseSquaredLR(LearningRateGetter):
    def get_learning_rate(self):
        """YOUR CODE HERE FOR Q4.2"""
        #raise NotImplementedError()
        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"""
        #raise NotImplementedError()
        self.num_evals +=1
        return self.multiplier/(self.num_evals ** 0.5)
```

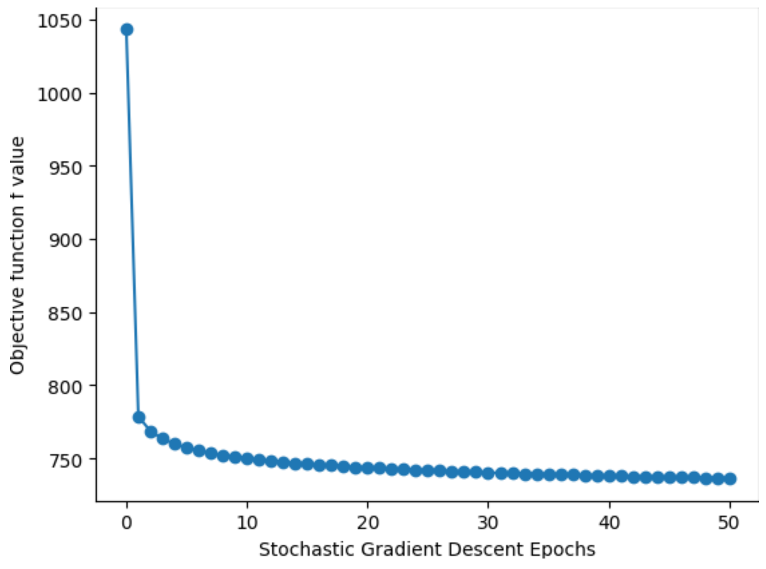
4.3 The Learning Curves (Again) [9 points]

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?

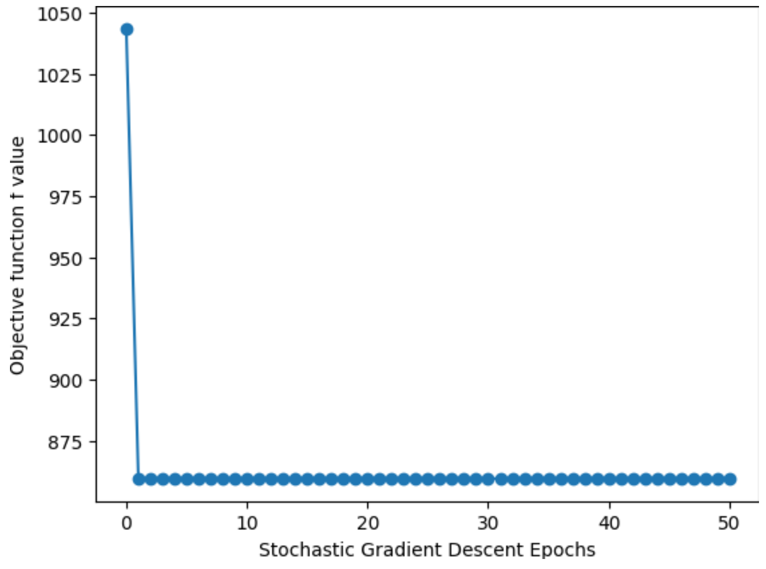
Answer:
Learning curve with constant LR:



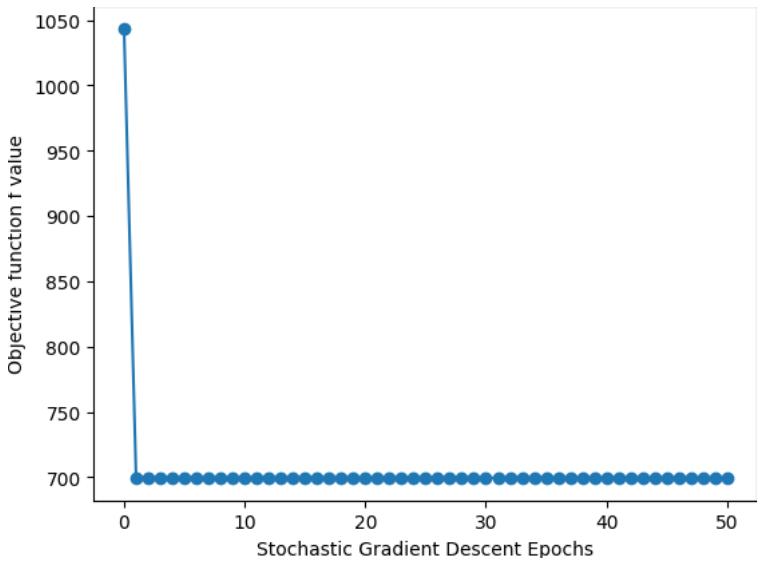
Learning curve with inverse LR:



Learning curve with inverse squared LR:



Learning curve with inverse square root of LR:



It seems like the step size of inverse square root of LR and inverse LR converge towards a global minimum.

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: It's faster if $n << d$.

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: It would allow k-means to work with non-convex clusters.

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

   Answer: MLE is the same thing as minimizing loss functions without regularization, whereas MAP is the same thing as minimizing loss functions with regularization.

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

   Answer: A generative model models X, whereas a discriminative does not model X.

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: In a kernel model, we can add 1 to the dot product that we calculate.

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

   Answer: No, because if we increase k, we would have more principle components, meaning that our reconstruction would resemble the original data set more and more. If we resemble the data set more, the loss should decrease, not increase.

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

   Answer: One of the uses of PCA is to reduce the dimensionality of our X dataset. If $k > d$, we would be increasing the dimensionality, which increases the computational cost.

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

   Answer: A single "eigenface" would be stored in W.

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

   Answer:
   An advantage is that stochastic gradient would run faster than SVD.
   A disadvantage is that while it runs faster, it still just give you an approximation, whereas SVD allows determine exactly what the Z matrix would be.