Assignment 9 - Feature learning & selection

This week, we'll go over some ways in which one may lower the amount of features used for training even without domain expertise, i.e., without being able to select them manually based on one's understanding of the relationships between the features and the label. There are multiple reasons one might want to use less features in training:

- to prevent overfitting (improving the ratio of training samples to features)
- to decrease computational complexity (faster training and prediction)
- visualizing high-dimensional data

In particular, we'll start off by looking at a method used even outside of machine learning called Principal component analysis (PCA) and use it to analyse and train on the MNIST dataset. Then we'll take a look at the Lasso and compare them. ### TODO: Possibly, At the end, we show parts of sklearn.feature_selection ###

Learning goals

After this assignment, you should

- understand why one might want to use a lower number of features
- understand PCA on an intuitive level
- be aware of the pitfalls of using PCA in machine learning
- know what is Lasso and how it relates to Ridge regression
- be able to use Lasso to select a subset of features

Principal Component Analysis (PCA)

You can think of Principal component analysis as finding a linear subspace of a given dimension (line, plane, hyperplane,...) that is the best fit of high-dimensional vectors. In this case, by best fit we mean that we minimize the squares of the distances of the vectors from the subspace, which corresponds to the information that we'll lose by this method.

If you recall how linear regression works, this may sound familiar. The difference is that linear regression studies functional dependence of the label on the features (how to fit a linear predictor), while PCA does not take the labels into account, it is only concerned about the features. The exact formulations are out of scope of this assignment but for visualizations, see for example this link (https://www.r-bloggers.com/2010/09/principal-component-analysis-pca-vs-ordinary-least-squares-ols-a-visual-explanation/).

There are a few pitfalls to PCA that you should be aware of:

- If we set the dimensions of the subspace too low, we may lose too much information to make any reasonable predictions.
- The features generated by PCA are a linear combination of the original features and there's likely no way to interpret them. Imagine a dataset where you try to predict car brand based on engine power and price. Using PCA to reduce the features to 1 might leave us with a feature "0.9*price-0.1*power", which doesn't have any real-world meaning.
- Since PCA doesn't take labels into account, it is entirely possible that the selected features will not be optimal for predicting the labels.
- PCA is very sensitive to statistical properties of the different dimensions (=features).

All of these will be illustrated below.

MNIST

We begin by fetching the MNIST dataset. This dataset contains 70000 B&W images with resolution 28x28 pixels. Each pixel represents a shade of grey as an 8bit integer (range 0-255) and each image represents a handwritten digit. This leads to a 10-class classification problem, where the goal is to predict the digit based on pixel intensities.

Below we fetch the datset and view one representative of each class to familiarize ourselves with the dataset.

In [1]:

```
# Import basic libraries needed for this assignment
%config Completer.use_jedi = False  # enable code auto-completion
import matplotlib.pyplot as plt
import numpy as np

import seaborn as sns

from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression, Lasso, Ridge, LinearRegress
ion
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.decomposition import PCA

# load the MNIST dataset
X_MNIST, y_MNIST = fetch_openml("mnist_784", version=1, return_X_y=True, as_fram
e=False)
```

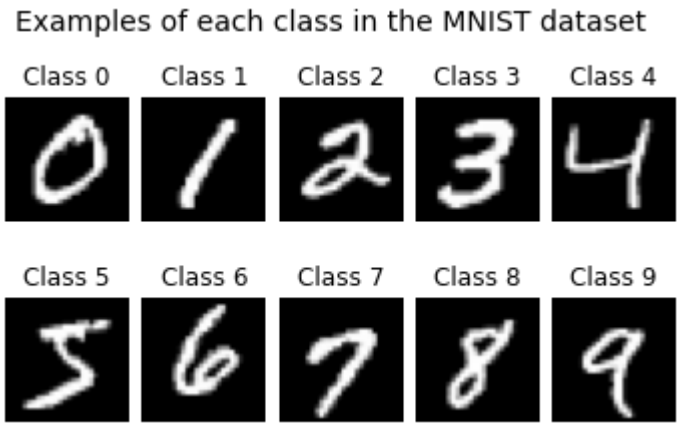
In [2]:

```
# plot the numbers in two rows
def plot_digits(X,y,cols=5,title=None):
    cols = 5
    fig, axs = plt.subplots(2, cols)
    plt.axis('off')
    for digit in range(10):
        # find the first representative of the label
        idx = np.argwhere(y == str(digit))[0]

        # change the vector into a 2D array and plot it
        im = X[idx].reshape(28, 28)
        axs[digit//cols, digit % cols].set_title(f'Class {digit}')
        axs[digit//cols, digit % cols].imshow(im, cmap='gray')
        axs[digit//cols, digit % cols].axis('off')

    # change the spacing between the subplots
    plt.subplots_adjust(wspace=0.1, hspace=-0.3)
    if title is not None:
        fig.suptitle(title,fontsize=14)
        fig.subplots_adjust(top=0.97)

plot_digits(X_MNIST,y_MNIST,title="Examples of each class in the MNIST dataset")
plt.show()
```



Demo: PCA, before and after

Below, we use Logistic Regression like we're used to, to predict the digits. Even though the images have a rather low resolution, training the model on more than a couple hundred images quickly becomes very computationally demanding and can take a couple of minutes. Performing a grid search to find the optimal parameters could thus easily take over an hour.

In [3]:

```
from sklearn.model_selection import train_test_split

# Split the dataset into a training set and a validation set
X_train, X_val, y_train, y_val = train_test_split(X_MNIST, y_MNIST, test_size=0.
33, random_state=42)

clf = LogisticRegression(solver='liblinear' )
clf.fit(X_train[:200], y_train[:200])
# can increase training data to reach 91.7 % but it takes 10 minutes on my compu
ter
```

Out[3]:

```
LogisticRegression(solver='liblinear')
```

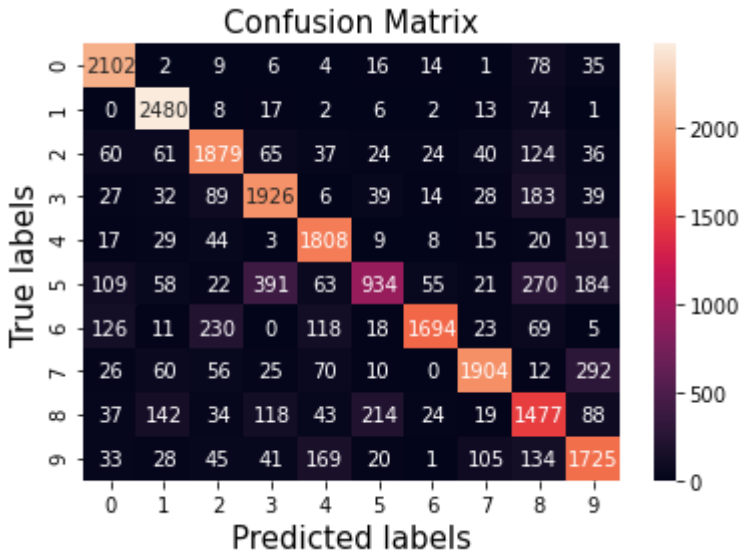
In [4]:

```
def generate_confusion_matrix(y_true, y_pred):
    # visualize the confusion matrix
    ax = plt.subplot()
    c_mat = confusion_matrix(y_true, y_pred)
    sns.heatmap(c_mat, annot=True, fmt='g', ax=ax)

    ax.set_xlabel('Predicted labels', fontsize=15)
    ax.set_ylabel('True labels', fontsize=15)
    ax.set_title('Confusion Matrix', fontsize=15)

y_pred = clf.predict(X_val)
generate_confusion_matrix(y_val, y_pred)
plt.show()

# compute the accuracy
multi_accuracy = accuracy_score(y_val, y_pred)
print(f"Prediction accuracy: {100*multi_accuracy:.2f}%")
```




```
Prediction accuracy: 77.61%
```

With PCA

First, we fit the PCA to the training data. By setting the `n_components` parameter, we can choose the dimension of the fitted subspace (=number of output features). A perhaps surprising result is that having computed the PCA, it is very simple to lower the amount of features further: the most important feature (in the PCA sense) is the first one, the most important two are the first two,...

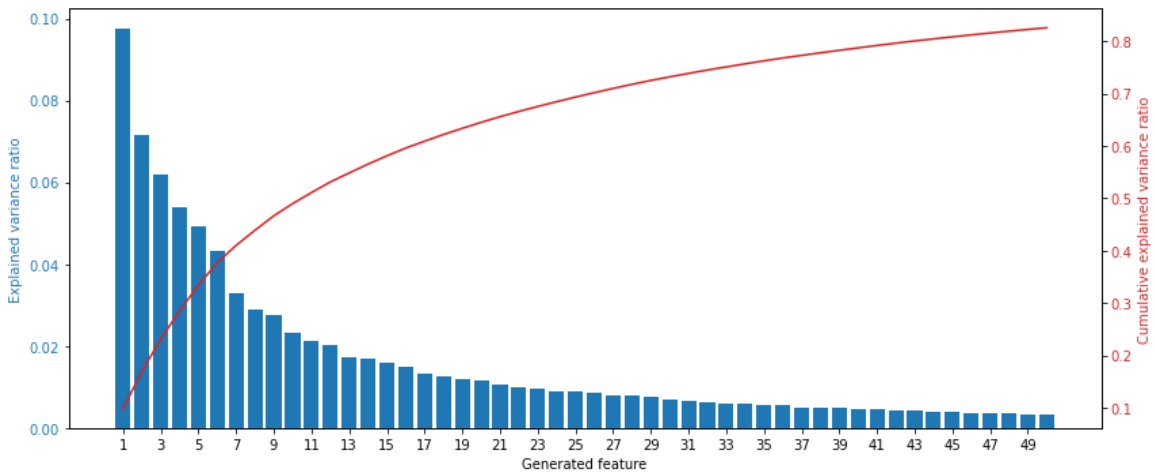
One way to decide how many components to use is to look at the explained variance ratio of each of the features. These correspond to the "importance" of each feature. Let's plot these:

In [5]:

```
# fit the PCA
N = 50
pca = PCA(n_components=N)
X_train_reduced = pca.fit_transform(X_train)

# plot the explained variances
fig, ax1 = plt.subplots(figsize=(12, 5))
color = 'tab:blue'
ax1.bar(1+np.arange(N), pca.explained_variance_ratio_, color=color)
ax1.set_xticks(1+np.arange(N, step=2))
ax1.tick_params(axis='y', labelcolor=color)
ax1.set_ylabel("Explained variance ratio", color=color)
ax1.set_xlabel("Generated feature")

ax2 = ax1.twinx()
color = 'tab:red'
ax2.tick_params(axis='y', labelcolor=color)
ax2.plot(1+np.arange(N), np.cumsum(pca.explained_variance_ratio_), color=color)
ax2.set_ylabel("Cumulative explained variance ratio", color=color)
fig.tight_layout()
plt.show()
```



If we choose to use 784 components, then the cumulative explained variance ratio will be 1, i.e., no data would be lost. This corresponds to projecting the data to the same space where it originally "lived", i.e. not doing anything. Note, however, that the features will likely still change, even though no data is lost. What happens is essentially a change of basis of the feature space.

Of course, there's not much use in keeping all the features, so how many features (components) should we keep? There's no clear cut answer to that here and there rarely is. In this case, let's keep 15 components, because the explained variance ratio drops to under 2% per feature around that point.

In [6]:

```
N = 15
pca.set_params(n_components=N)
X_train_reduced = pca.fit_transform(X_train)
```

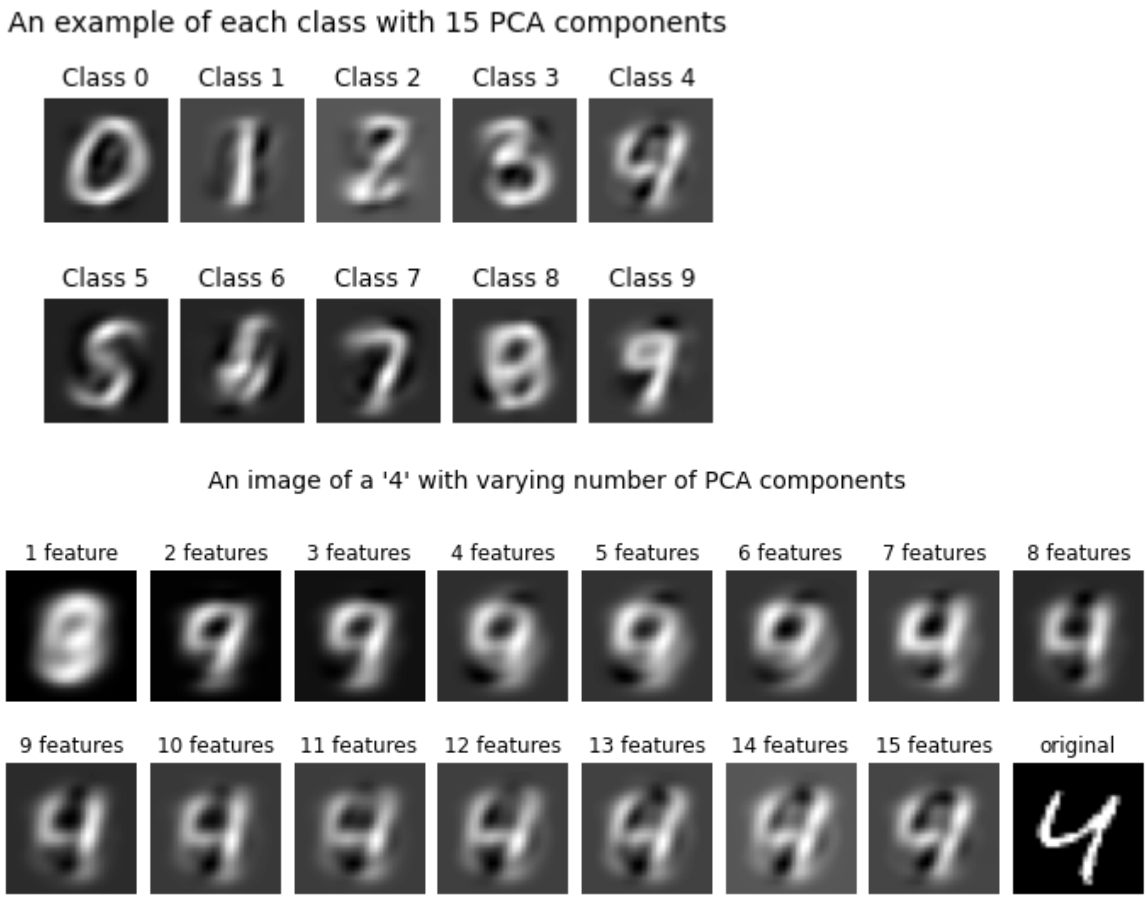
Now let's visually confirm that the compression hasn't affected the pictures too much. Based on the representatives chosen, it seems that keeping just 15/784 dimensions still leaves the digits legible, if slightly blurry.

To view the image after dropping the less important features, we need to transform the 15-dimensional vector back into the original 784-dimensional space. For this, we'll use the `inverse_transform` method. Mathematically, all of the components form a basis of the original space, so this function creates a linear combination of the first 15 basis vector with coefficients corresponding to the new features.

In [7]:

```
# plot the digits with only 15 components
plot_digits(pca.inverse_transform(X_train_reduced), y_train, title="An example o
f each class with 15 PCA components")
plt.show()

# plot the same picture with different numebrs of features
digit = 4
cols = 8
fig, axs = plt.subplots(2, cols, figsize=(12, 8))
fig.suptitle("An image of a '4' with varying number of PCA components", fontsize
=14)
idx = np.argwhere(y_train == str(digit))[0]
for n in 1+np.arange(N):
    X_nfeatures = X_train_reduced[idx]*[1 if i < n else 0 for i in range(N)]
    # transform the data back into the original 784-dimensional space
    im = pca.inverse_transform(X_nfeatures).reshape(28, 28)
    # plot it
    axs[(n-1)//cols, (n-1) % cols].set_title(f"{(n)} feature{'s' if n>1 else ''}
")
    axs[(n-1)//cols, (n-1) % cols].imshow(im, cmap='gray')
    axs[(n-1)//cols, (n-1) % cols].axis('off')
# plot the original
axs[-1, -1].set_title(f'original')
axs[-1, -1].imshow(X_train[idx].reshape(28, 28), cmap='gray')
axs[-1, -1].axis('off')
# adjust spacing
fig.subplots_adjust(wspace=0.1, hspace=-0.8, top=1.3)
plt.show()
```

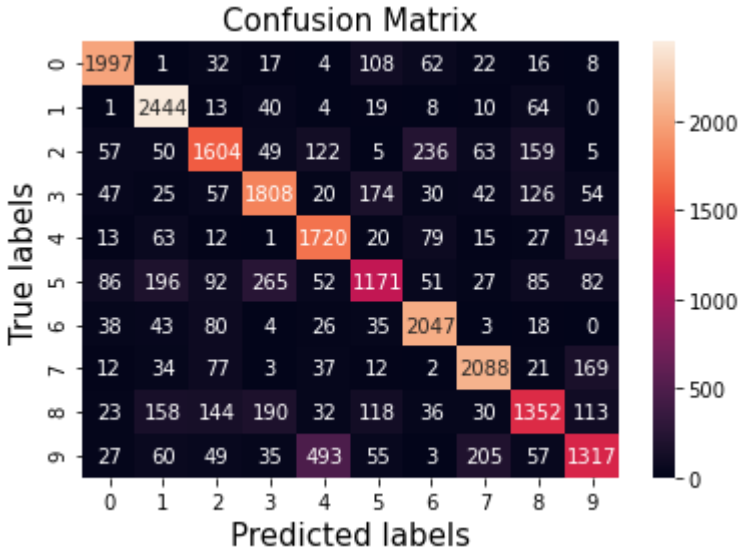


Student Task A9.1

Use the transformed data `X_train_reduced` to train a logistic regressor. Then compute its accuracy on the validation set. Do not forget what variables have already been transformed using PCA and which still need to have it applied.

In [13]:

```
Prediction accuracy: 75.97%
```



In [14]:

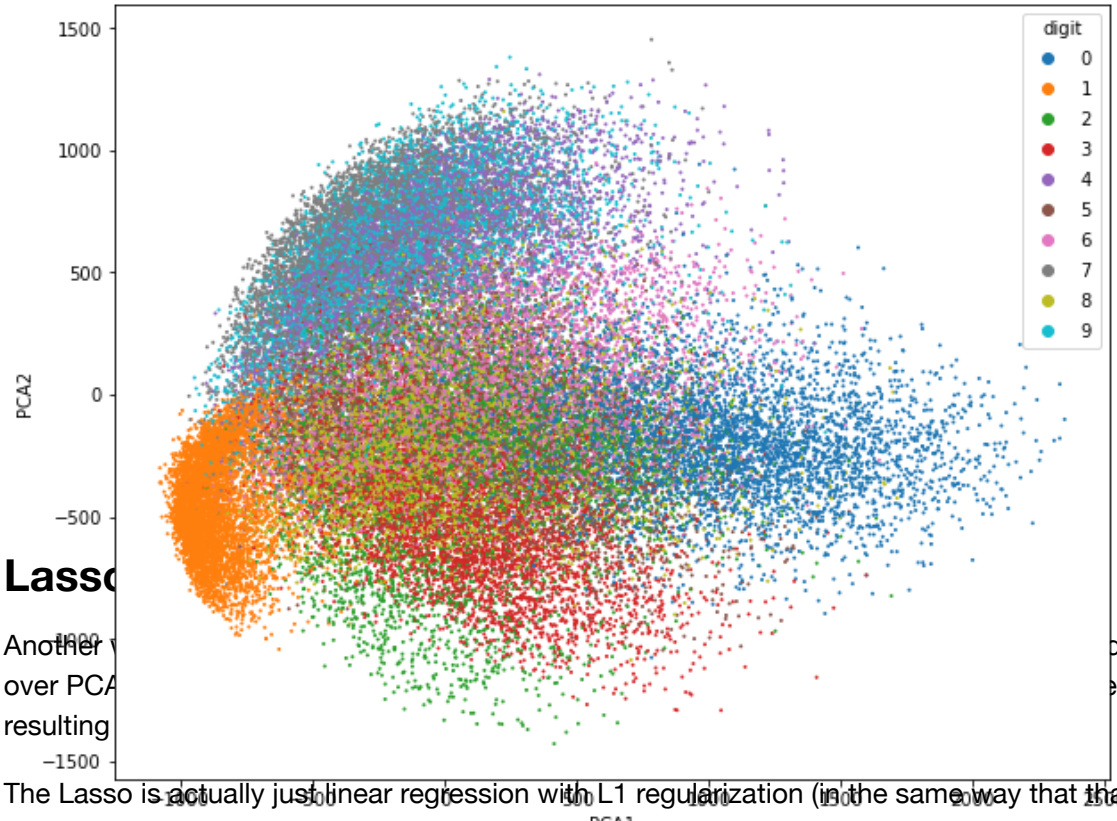
In [15]:

Demo: Data visualization

Another application of PCA is visualization of high-dimensional data. Here, we plot each image as a single point based on its coordinates in the 2D subspace generated by PCA. For example, we can see that 1's seem to be very easy to tell apart and they all look very similar, whereas these two dimensions are certainly not enough to tell apart 4's and 9's in any meaningful way. This confirms what we saw earlier - with only a few components, the 4 looked like a 9!

Note that since the features carry no meaning anymore, we label the axes simply as PCA1 and PCA2.

In [16]:



Lasso

Another v dvantage it has over PCA etability of the resulting

The Lasso is actually just linear regression with L1 regularization (in the same way that that in A6 we used Ridge - L2 regularization). That means that the squared error loss is replaced by

$$L((\mathbf{x}, y), h^{(\mathbf{w})}) = (y - \mathbf{w}^T\mathbf{x})^2 + \lambda\|\mathbf{w}\|_1.$$

By varying parameter λ, we influence the number of features that will have zero weights in the resulting trained model.

The reason why using L1 regularization would typically lead to sparse ("many zeros") solutions is not immediately clear. The animation below (courtesy of Itay Evron (https://github.com/ievron/RegularizationAnimation/)) could give you an intuition (along with the description underneath):

The unit "circles" (points with norm ≤ 1) are shown in turquoise. Notice that in the 1-norm case (on the right), the unit circle is actually a square rotated by 45 degrees. Convince yourself this is indeed the case when

$$\|(x, y)\|_1 = |x| + |y|.$$

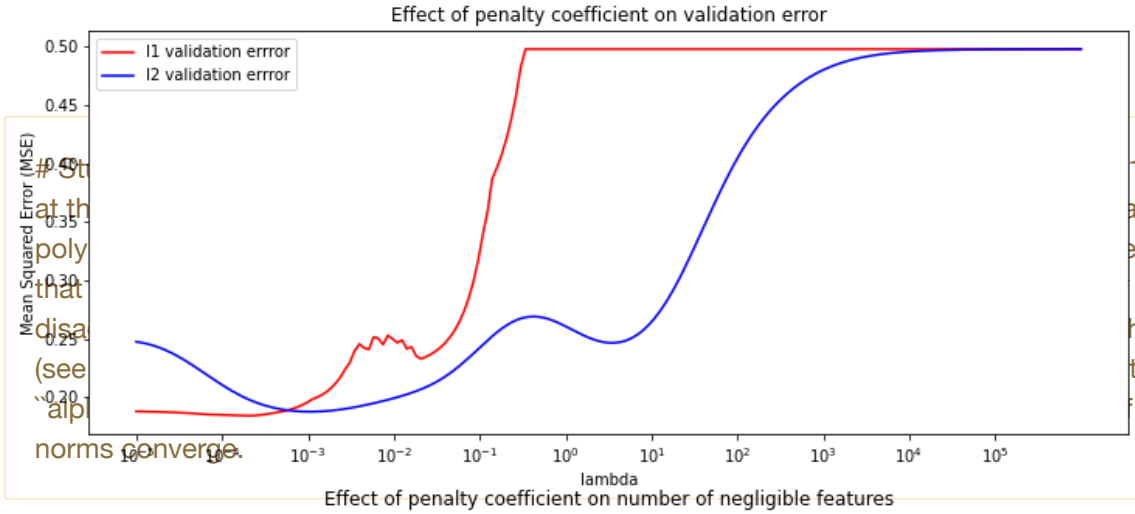
In grey, you can see the optimal value of the loss function (the center of the ellipses), each of the ellipses marks a countour line (curve where the loss assumes the same value). As the loss function changes (this corresponds to changing the training data), you can watch how does the orange point, which is the optimum on the unit circle, behaves. You can notice that in the case of L1 regularization, it is much more often on one of the axes, meaning that one of the weights is zero.

Demo: Comparison of Lasso and Ridge

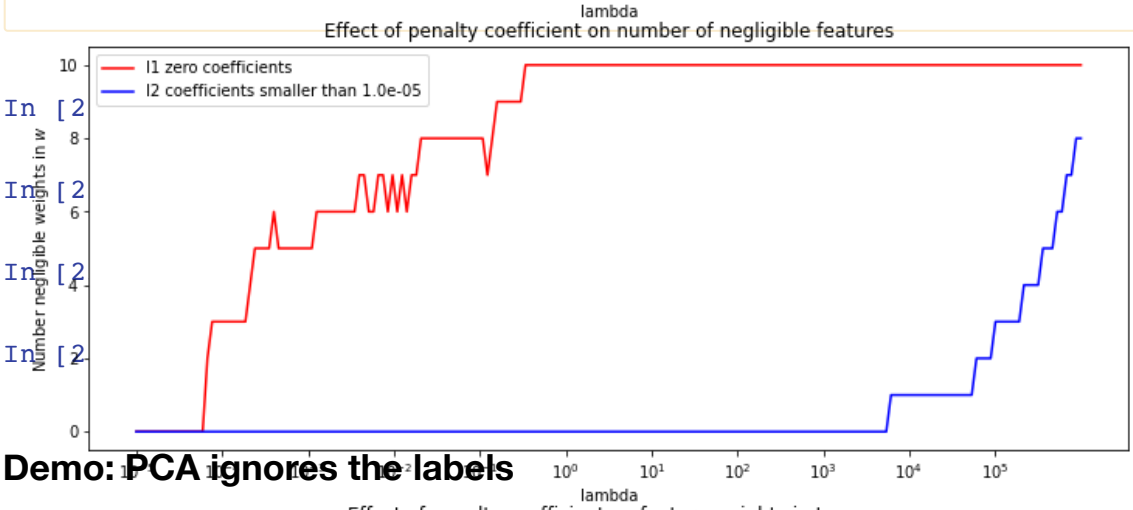
Now let's study how Lasso compares to Ridge. The dataset we use for training is the same we used when we studied ridge regression originally (in A6): noisy sinusoidal data. We will use polynomial features of degree 10 and look at how the penalty coefficient affects the resulting model.

The first plot compares how the error depends on parameter `alpha` and is not of very high importance here. In the plots that follow, you can observe how increasing the penalty weight, lasso tends to decrease the importance of features one at a time, while with ridge, this affects all features mostly similarly. Also

In [17]:



St 1: By looking
at th a degree 10
poly e, it is certain
that `A2=0`:
disa high `alpha`
(see tion as
`alp f `alpha`, the
norms converge.



In [2

In [2

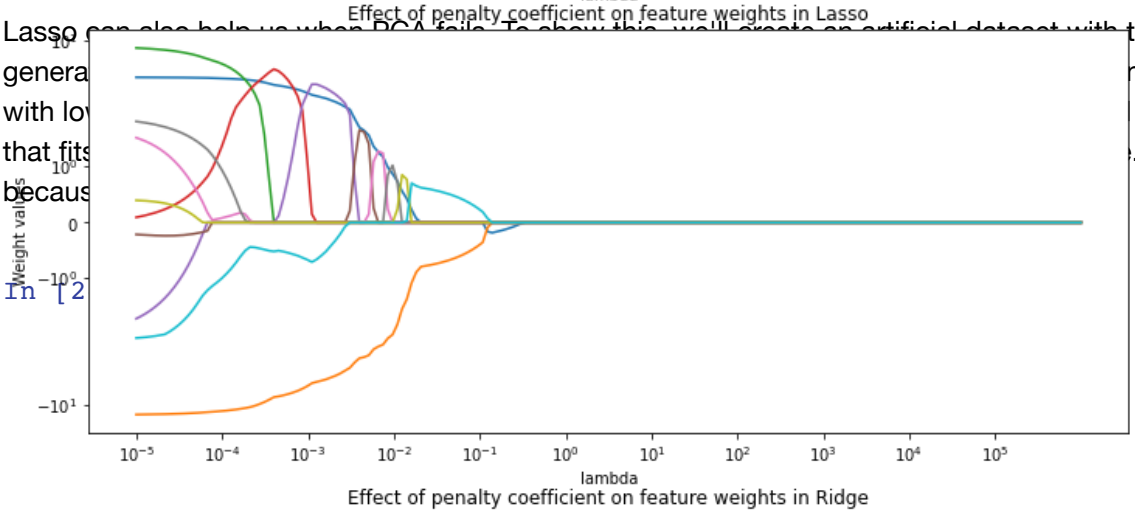
In [2

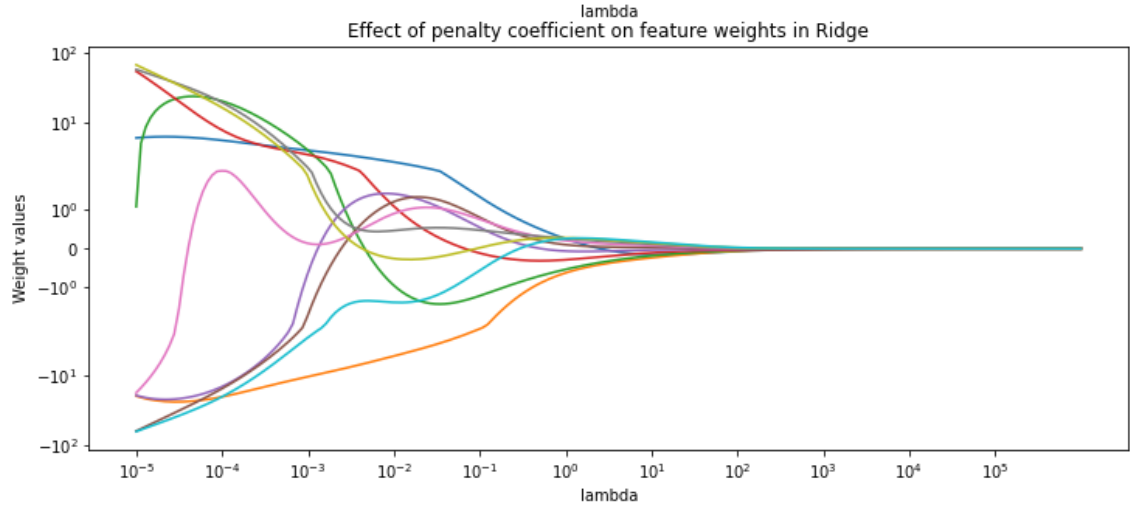
In [2

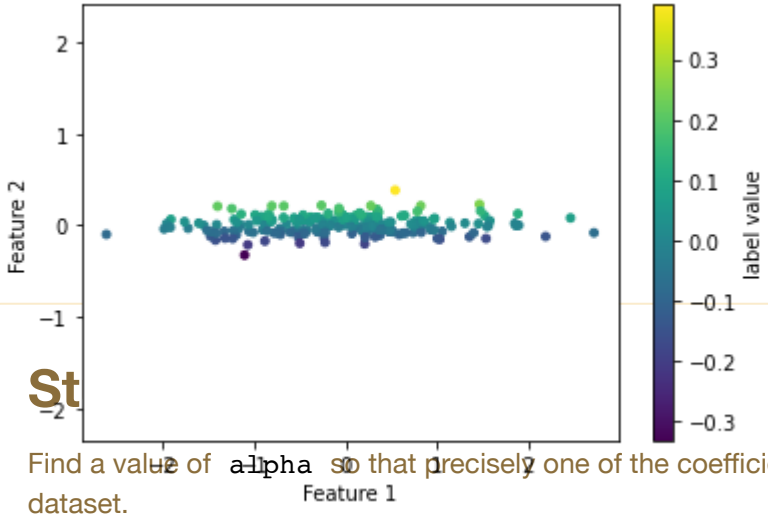
Demo: PCA ignores the labels

Lasso can also help us when PCA fails. To show this, we'll create an artificial dataset with two randomly
genera n the feature
with lo a 1D subspace
that fits . This is
becaus

In [2







St

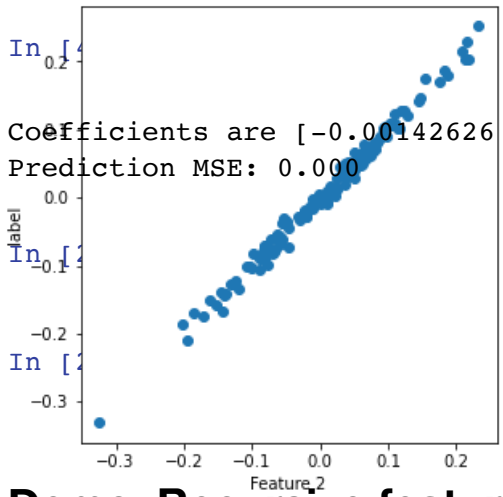
Find a value of `alpha` so that precisely one of the coefficients is nonzero after training on this dataset.

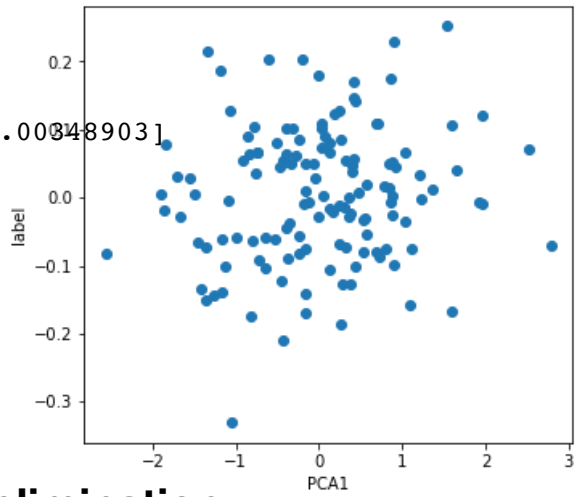
```
In [
```

```
Coefficients are [-0.00142626  1.00348903]
Prediction MSE: 0.000
```

```
In [
```

```
In [
```





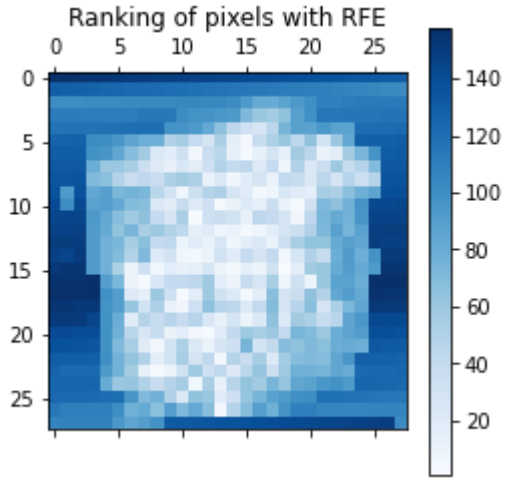
Demo: Recursive feature elimination

There are also many methods for [feature selection (https://scikit-learn.org/stable/modules/feature_selection.html)](https://scikit-learn.org/stable/modules/feature_selection.html) already implemented in `sklearn`. Here, we will take a quick look at Recursive feature elimination (RFE).

Given an external estimator (e.g. Logistic Regression), RFE iteratively drops the "least important" feature(s) after fit. In the case of Logistic Regression, the feature(s) dropped will be the feature(s) corresponding to the weight(s) that has (have) the smallest absolute value.

Below, you can see that according to RFE, the most important pixels seem to be those in the middle of the image, which is rather unsurprising - in this case, this is clear to a human that this would be the case. This approach can thus be useful when we do not have a good understanding of the relationships between the features and labels.

```
In [28]:
```


Ranking of pixels with RFE
0
5
10
15
20
25
140
120
100
80
60
40
20