Assignment 1: Linear Regression & KNN

Ash

2018/12/06

Moore's Law

Extract the date and base speed from the data

```
#Loading the data
data = pd.read_csv('/Users/ash/Downloads/benchmarks.csv')

#Extracting the date
data["cpu"], data["date"], data["somestuff"] = data['testID'].str.split('-'
    , 2).str
data["date"] = pd.to_datetime(data["date"], yearfirst=True)
data.dropna(subset=['date'], inplace=True)
```

Since there are some entries without date to extract from and there are only a few of them (578 missing values for some cpus in the 90s, compared to the total of 136995 entries - only 0.4%) I decided to drop these entries from analysis.

Semi-log plots

There are 73 benchmarks in total, some of which were only used for a specific period. Just to get a feel of different benchmarks, we plot a few of them in figure 1 and figure 2 in log_2 plots.

```
uniques = data['benchName'].unique()
m = 4
n = 1
nplot = m*n
count = 1
for _ in uniques[:nplot]:
    many_cpus = data[data['benchName'] == _]
    x = np.array(many_cpus['date'], dtype='datetime64[us]')
    y = np.log2(many_cpus['base'])
    plt.subplot(m,n,count)
    plt.scatter(x, y, s=3, label=_)
    plt.legend(loc=2)
    plt.xlabel('Time')
    plt.ylabel('$log_{2}$ of base speed')
    count += 1
plt.show()
print(data.groupby('benchName').base.nunique().sort_values())
```

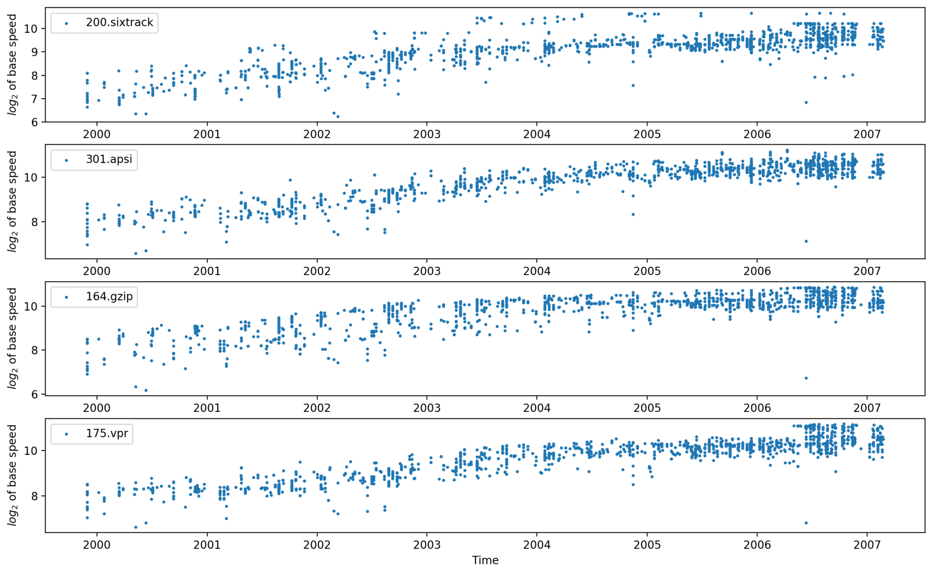


Figure 1: Semi-log of base speed for some benchmarks - 1

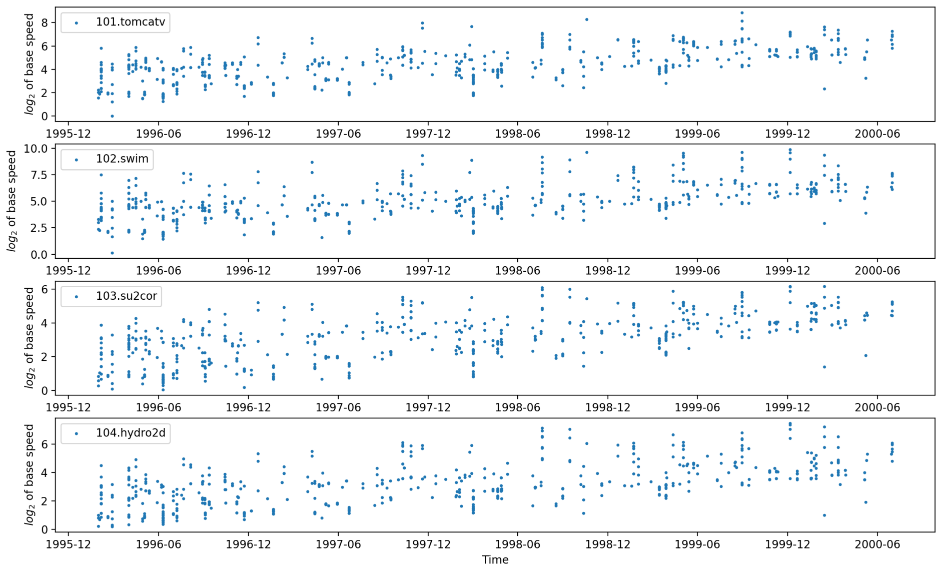


Figure 2: Semi-log of base speed for some benchmarks - 2

Also plotting the semi-log graph for "178.galgel" in figure 3. This benchmark was chosen because it has the most base speed data.

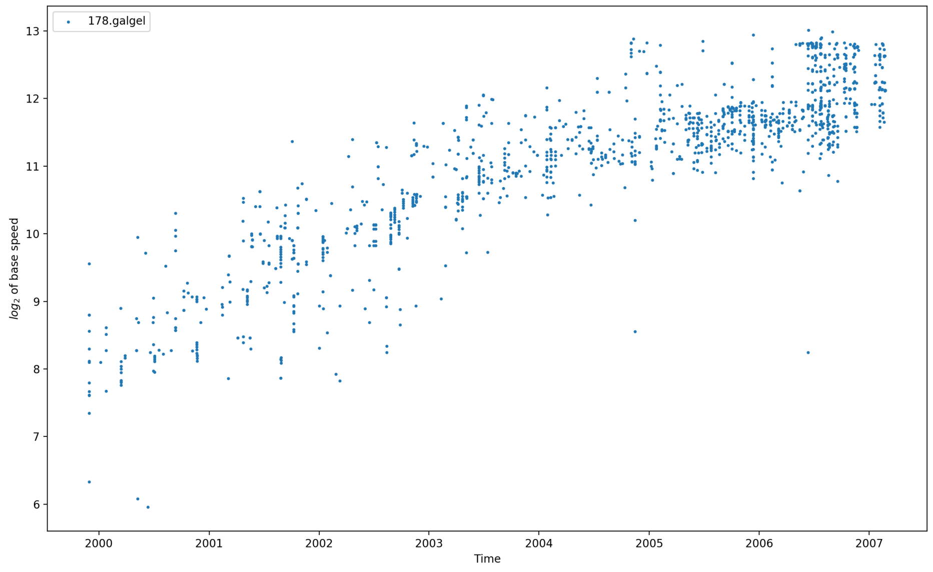


Figure 3: Semi-log of base speed for 178.galgel

Fitting linear models

To fit a linear model for the log of base speed of each year (since we are examining Moore's law) we need to average the data for each year because ordinary least square will minimize the mean squared error between a linear regression form relating the year to the log of base speed:

$$base = \theta_1 year + \theta_0 \qquad (or \;\; y = \theta_1 x + \theta_0)$$

which means minimizing:

$$error = (y_{truth} - y_{pred})^2 = (y_{truth} - \theta_1 x - \theta_0)^2$$

Linear regression is convenient because it has a closed form solution for its coefficients, and it also has a nice statistical interpretation: if we assume that the likelihood function (distribution of θ over the given data) and the prior (distribution of y and x themselves) are normally distributed, and all data points are identically and independently distributed (i.i.d.), we can consider the linear regression problem as updating the prior to find the posteriors for the whole dataset, which returns the maximum likelihood result of the parameters θs).

Fitting a linear regression by using scikitlearn for the averages of 178.galgel:

```python
many_cpus = data[data['benchName'] == '178.galgel']
many_cpus['year'] = many_cpus['date'].dt.year
x = [_ for _ in range(many_cpus['year'].min(), many_cpus['year'].max())]
y = []
for _ in range(many_cpus['year'].min(), many_cpus['year'].max()):
    dumb = many_cpus[many_cpus['year'] == _]
    y.append(dumb['base'].median())
x = np.array(x).reshape(-1, 1)
y = np.log2(y)

model = linear_model.LinearRegression()
model.fit(x, y)
print(model.coef_)
print(model.intercept_)
pred = model.predict(x)
plt.scatter(x, y)
plt.plot(x, pred, color='red')
plt.xlabel('Time')
plt.ylabel('$log_{2}$ of base speed')
plt.show()
```

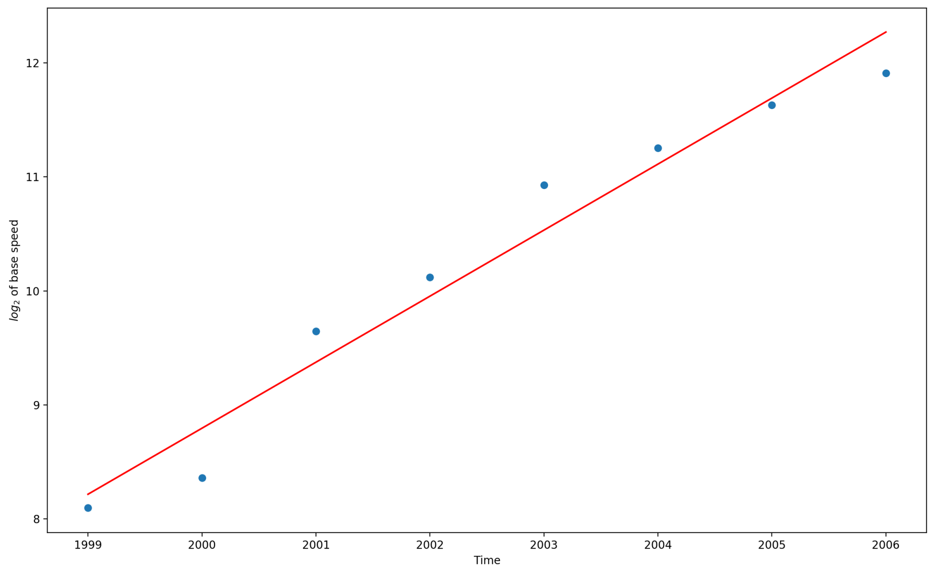


Figure 4: Semi-log of base speed for 178.galgel with best fit line ($\theta_1 = 0.5787779$)

We can also do linear regression models for all the available benchmarks like in figure 5. Eye-balling the graph we can see that generally the slope of the best fit lines are similar, but by some more recent benchmarks the slope is steeper suggesting a more rapid growth of computational power.

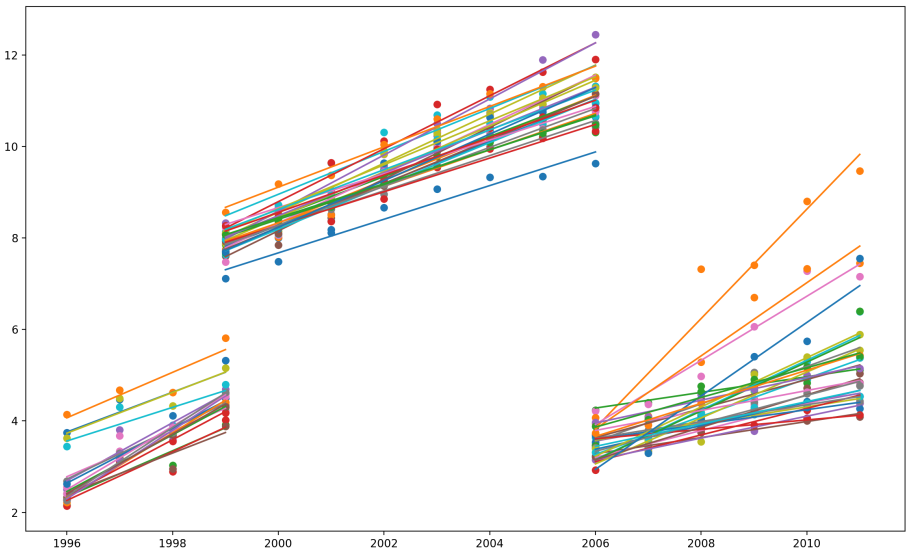


Figure 5: Semi-log of base speed for all benchmarks with best fit lines

Moore's law

Assume at year 0 the computational power (expressed through base speed in this case) is a. If Moore's law is right, then the base speed will double every year, which means:

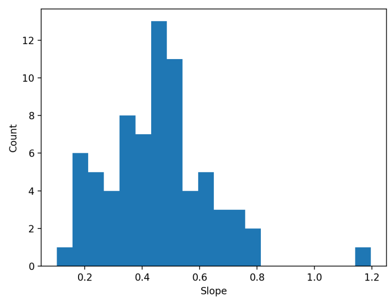
$$base_{year_n} = 2^{\frac{n}{2}} base_{year_0} = 2^{\frac{n}{2}} a$$

or

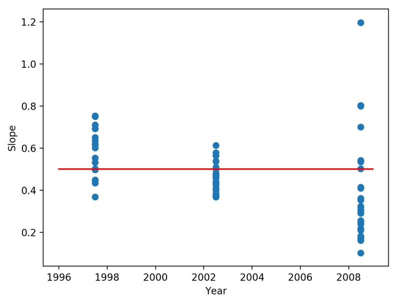
$$log_2(base_{year_n}) = log_2\left(2^{\frac{n}{2}}\right) + log_2(a) = 0.5n + log_2 a$$

so the coefficient (slope) of the best fit line for the semi-log plot will be roughly 0.5 if it obeys Moore's law. For the 178.galgel benchmark we can see that the slope is 0.58, which means it grew faster that Moore predicted. However this is just one benchmark and if we want a closer picture we need to investigate further. If we fit a linear regression models for all 73 benchmarks and extract their slopes:

```python
def test_moore(cpu, benchmark):
    many_cpus = cpu[cpu['benchName'] == benchmark]
    many_cpus['year'] = many_cpus['date'].dt.year
```



(a) Histogram of all benchmark slopes for best fit lines



(b) Graph of all benchmark slopes for best fit lines over year

Figure 6: Slopes of all benchmark

```
    x = [_ for _ in range(many_cpus['year'].min(), many_cpus['year'].max())
    ]
    y = []
    for _ in range(many_cpus['year'].min(), many_cpus['year'].max()):
        dumb = many_cpus[many_cpus['year'] == _]
        y.append(dumb['base'].median())
    x = np.array(x).reshape(-1, 1)
    y = np.log2(y)
    model = linear_model.LinearRegression()
    model.fit(x, y)
    return model.coef_


uniques = data['benchName'].unique()
coef_ = []
for _ in uniques:
    coef_.append(test_moore(data,_))

print(np.mean(coef_))
print(np.median(coef_))
plt.hist(np.array(coef_), bins=20)
plt.show()
```

After running the above script we will find that the mean of the benchmark growth rates is about 0.4565 and the median is about 0.4493, which is quite close to Moore's law (in figure 6a the histogram of the slopes also concurs). However if we plot the slopes by its time of being used (figure 6b) we will see that in 2008 the slops are more spread out comparing to the other two time marks, suggesting that benchmarks used in 2008 is more inconsistent in terms of its growth of base speed over time, and fewer benchmarks are above the 0.5 red line.

MNIST Digits

Load MNIST and plot examples

Load the abridged version of the MNIST dataset by scikitlearn then plot some examples (presented in figure 7):

```python
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
import matplotlib

#Plotting function
def plot_digits(digits , m, n):
    ndigits = m*n
    count = 1
    for _ in range(ndigits):
        plt.subplot(m,n,count)
        im = digits.images[_]
        plt.imshow(im, cmap=matplotlib.cm.binary,
                   interpolation="nearest")
        plt.axis("off")
        plt.title(str(digits.target[_]))
        count += 1
    plt.show()

#Loading and plotting some examples from dataset
mnist = load_digits()
plot_digits(mnist, 5, 3)
```

KNN for two classes and error rate

KNN is a non-parametric machine learning algorithm that is used for classification of multiple classes by measuring distances between a new instances and its "neighbors" in the high-dimensional feature space and classify the new instances by a majority vote (or a weighted vote) of the classes of its neighbors. Probabilistically, KNN is assigning each training instance a Gaussian distribution of variance σ^2 in the feature space then model each class by a mixture of Gaussians. For a new instance, the probability of it being in each class is considered the probability value of those mixture of Gaussians in the specific place that new instance is located in the feature space. As σ^2 approach 0, only the nearest neighbor of the new instance has an effect on its classification (since only with the nearest neighbor class the new instance has any probability mass because other Gaussians variance shrink arbitrarily close to 0) but if $\sigma^2 \not{0}$ then other neighbors has some effects on the new instance classification. Using KNeighborsClassifier class in scikitlearn we build the KNN model for the small MNIST dataset with a binary classification of 2 or 5:

```python
#Loading Model
```

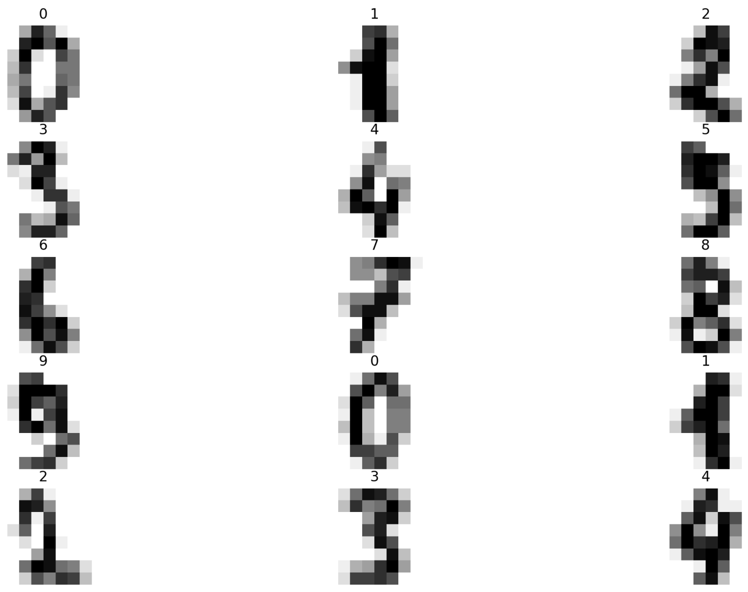


Figure 7: Small MNIST examples

```
k = 10
weights = 'distance'
knn = KNeighborsClassifier(k, weights=weights)

# #Choosing two numbers to build model
first = 2
second = 5

X_train_1st = mnist.data[mnist.target == first]
y_train_1st = mnist.target[mnist.target == first]
X_train_2nd = mnist.data[mnist.target == second]
y_train_2nd = mnist.target[mnist.target == second]
X = np.concatenate((X_train_1st/255, X_train_2nd/255), axis=0)
y = np.concatenate((y_train_1st, y_train_2nd), axis=0)

#Shuffling to prevent any artificial sequential order
shuffling = np.arange(X.shape[0])
np.random.shuffle(shuffling)
X = X[shuffling]
y = y[shuffling]

#Train/test split
X_train = X[:math.floor(0.8*X.shape[0]),:]
X_test = X[math.floor(0.8*X.shape[0]):,:]
y_train = y[:math.floor(0.8*X.shape[0])]
```

```python
y_test = y[math.floor(0.8*X.shape[0]):]


knn.fit(X_train,y_train)
pred = knn.predict(X_test)
acc = [pred == y_test]
print('Accuracy:', sum(acc[0])/y_test.shape[0])
```

In this model we get an accuracy of 100% because it a relatively small dataset and we have a high-dimensional feature space ($8 \times 8 = 64$ features). Let us test the KNN model with multi-class classification:

```python
#Shuffling to prevent any artificial sequential order
X = mnist.data
y = mnist.target
shuffling = np.arange(X.shape[0])
np.random.shuffle(shuffling)
X = X[shuffling]
y = y[shuffling]

#Train/test split
X_train = X[:math.floor(0.8*X.shape[0]) ,:]
X_test = X[math.floor(0.8*X.shape[0]): ,:]
y_train = y[:math.floor(0.8*X.shape[0])]
y_test = y[math.floor(0.8*X.shape[0]):]


knn.fit(X_train,y_train)
pred = knn.predict(X_test)
acc = [pred == y_test]
print('Accuracy:', sum(acc[0])/y_test.shape[0])
```

The accuracy for 10 classes is around 98.33% which is pretty impressive for such a simple model. Still, this is a smaller version of MNIST, and in practice KNN is rarely used with dataset that has either a large number of features or many instances in the training set because KNN needs to store the whole training set to make inference, which is very memory-costly.

Full MNIST

Loading the full MNIST dataset (60,000 training examples of 28x28 images for 10 classes) with keras, we build the same KNN model for binary classification of 1 or 3:

```python
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#Loading Model
k = 10
weights = 'distance'
knn = KNeighborsClassifier(k, weights=weights)

# #Choosing two numbers to build model
```

```python
first = 1
second = 3

X_train_1st = x_train[y_train == first]
y_train_1st = y_train[y_train == first]
X_train_2nd = x_train[y_train == second]
y_train_2nd = y_train[y_train == second]
X = np.concatenate((X_train_1st/255, X_train_2nd/255), axis=0)
y = np.concatenate((y_train_1st, y_train_2nd), axis=0)


X_test_1st = x_test[y_test == first]
y_test_1st = y_test[y_test == first]
X_test_2nd = x_test[y_test == second]
y_test_2nd = y_test[y_test == second]
X_test = np.concatenate((X_test_1st/255, X_test_2nd/255), axis=0)
Y_test = np.concatenate((y_test_1st, y_test_2nd), axis=0)

#Shuffling to prevent any artificial sequential order

shuffling = np.arange(X.shape[0])
np.random.shuffle(shuffling)
X = X[shuffling]
y = y[shuffling]


knn.fit(X.reshape(X.shape[0], 28*28),y)
pred = knn.predict(X_test.reshape(X_test.shape[0], 28*28))
acc = [pred == Y_test]
print('Accuracy:', sum(acc[0])/Y_test.shape[0])
```

It took around 4 minutes for my machine to build the KNN model and make predictions and achieved an impressive 99.94% accuracy on the test set, but again this feature space has 28x28 dimensions so it's possible that in this high-dimensional space these two digits is just separated really distinctively. To test this hypothesis we will use principle component analysis to reduce the dimension of the original to 2 and 3 and plot them to see if they are really separated in these spaces:

```python
from sklearn.decomposition import PCA
pca = PCA(n_components=2)

new_X = pca.fit_transform(X_test.reshape(X_test.shape[0], 28*28))
print(pca.explained_variance_ratio_)
X_2 = new_X[Y_test == 1]
X_5 = new_X[Y_test == 3]

plt.scatter(X_2[:,0], X_2[:,1], color='red')
plt.scatter(X_5[:,0], X_5[:,1], color='blue')
plt.show()
```

In 2 dimensions, even though the explained variance is only 21% and 12% for the 1st and 2nd dimension respectively, we can see in figure 8 that they are quite separated. In 3 dimensions, the 3rd dimension only adds a 5% explained variance but we can see that they look even more separated. It is very likely that in high dimensional space (where 62% of the remaining variance is located) they are clustered in completely different groups, and therefore the KNN model was able to predict them with such high accuracy.

```
pca = PCA(n_components=3)

new_X = pca.fit_transform(X_test.reshape(X_test.shape[0], 28*28))
print(pca.explained_variance_ratio_)
X_2 = new_X[Y_test == 1]
X_5 = new_X[Y_test == 3]

ax = plt.axes(projection='3d')
ax.scatter3D(X_2[:,0], X_2[:,1], X_2[:,2], 'red')
ax.scatter3D(X_5[:,0], X_5[:,1], X_5[:,2], 'red')
plt.show()
```

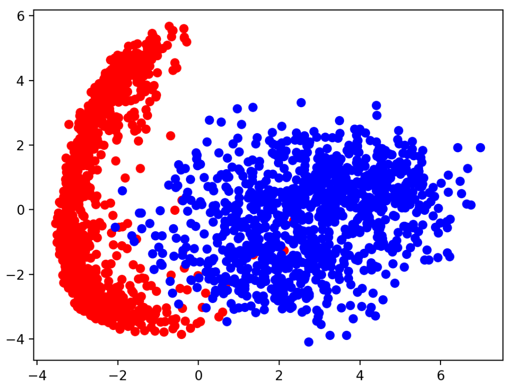


Figure 8: 1s and 3s of the original MNIST in 2 dimensions, reduced by PCA

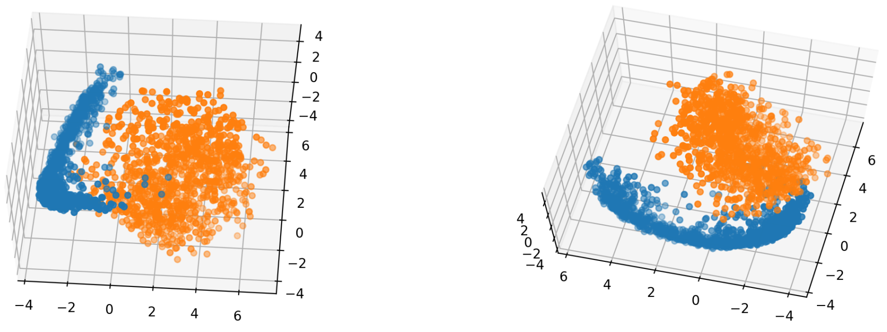


Figure 9: 1s and 3s of the original MNIST in 3 dimensions, reduced by PCA