Machine Learning A - Exam

Exam Number 135

November 2021

1 Alzheimer's disease diagnosis

1.1

To open and inspect the data I used Numpy as follows:

```
train_input = np.genfromtxt("data/trainInput.csv",delimiter = ",")
```

Again using Numpy, I could get the counts and frequencies this way:

```
n, c = np.unique(train_target, return_counts = True)
print("Frequencies:", c/len(train_target))
```

Which gives about 0.626 for the first class and 0.374 for the second. This number is consistent between the training and test sets.

1.2

To perform the Principal Component Analysis I used the scikit-learn package from sklearn.decomposition.PCA. After consulting with both the book and the associated online lectures[1] I choose to center the inputs before the PCA but not normalize, as we in the following questions are interested in the variance of the original features.[2] In code, all that was needed was the following lines:

```
from sklearn.decomposition import PCA
pca = PCA()
cent = (train_input - train_input.mean(axis = 0))
pca.fit(cent)
```

I could then extract the eigenvalues by the following command:

```
eigenvalues = pca.explained_variance_
```

[1] Very recommendable `https://www.youtube.com/watch?v=FdqOpFVp25M&t=1s`

[2] As the co-author Malik Magdon-Ismail says in the above lecture: "If you are agnostic about the original data you should normalize, but if you do care about the scale, you don't scale"

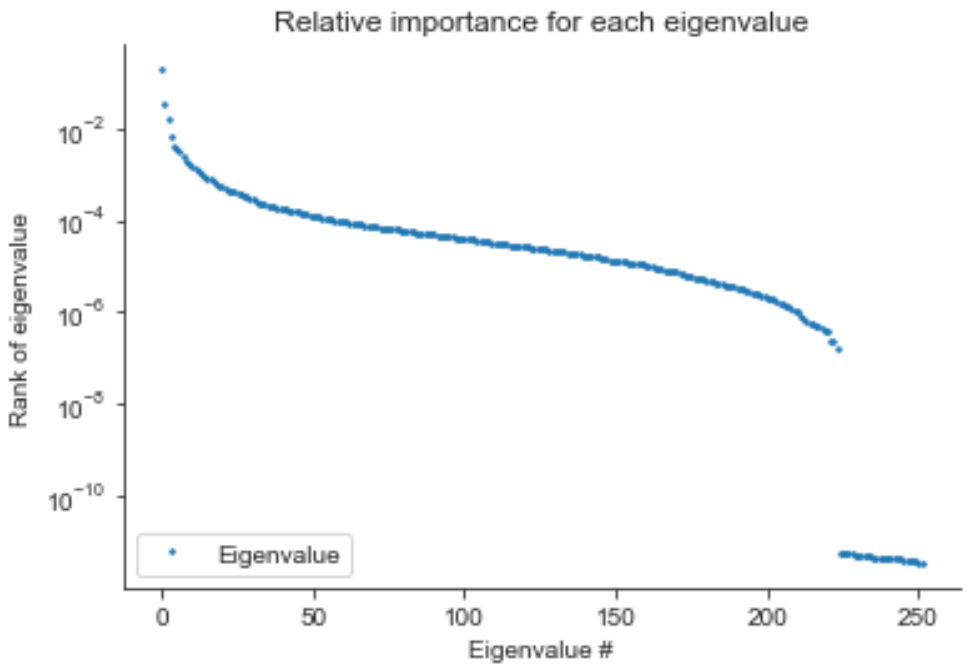

Figure 1: A visualization of the eigenspectrum of the PCA. I chose a logarithmic y-axis as the first and the last eigenvalues are multiple orders of magnitude apart.

I choose to use the "pca.explained_variance_" as this corresponds to the eigenvalues from the computed data/empirical covariance matrix while also being scaled by Bessel's correction. The higher the rank, the more variance is explained by the corresponding eigenvector. A plot of the eigenvalues can be seen in figure 1.
To find how many eigenvalues was needed to explain 90% of the variance I wrote this code snippet:

```
def find_90():
    cumvar = np.cumsum(pca.explained_variance_ratio_)
    for n, cu in enumerate(cumvar):
        if cu >= 0.9:
            return n
```

This returned 6, meaning the top 7^3 eigenvalues can explain 90% or more of the variance. In reality, they explain about 90.054%. This can also be seen visually in figure 2.

[3]This is not an exponent. I just meant to say, that is because Python is indexing from 0 of course.

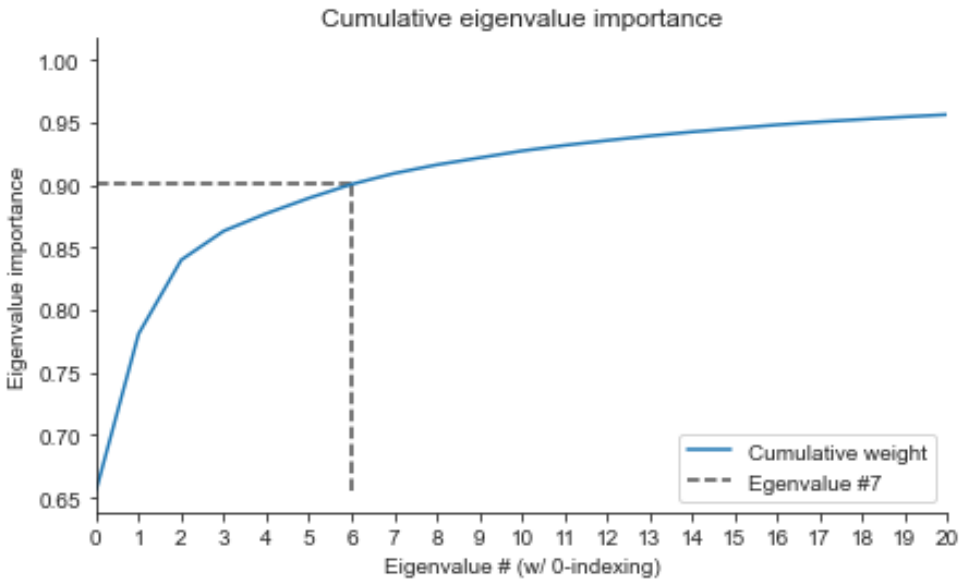

Figure 2: Here it can be seen, that the top 7 eigenvalues are needed to explain 90% of the variance. For clarity the eigenvalues are cut off at 20.

To visually see the data points and their split, they have been projected down onto the two most important PCA-axes and marked according to their class. This can be seen in figure 3.

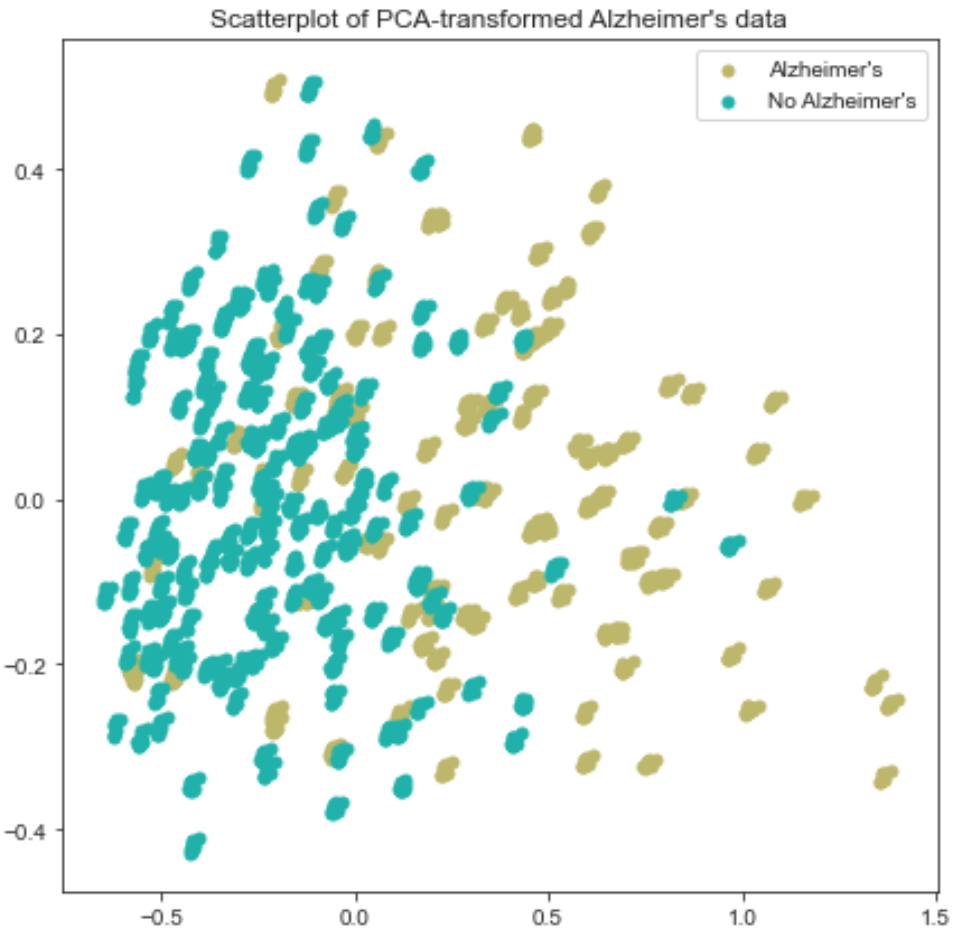

Figure 3: The data point in a 2D scatter plot using the two most important PCA-components as the x- and y-axis. For the labels I have assumed that there was more data points of people without Alzheimer's than with.

A note: I believe it can be seen from the 'clumps' in the data, that there has been a projection from a higher dimensional space where the points lie close to each-other within each class. Spoiler alert: This might also explain the quality of the low-k k-means clustering in the next question where we use the full dimensionality.

1.3

Now, to do k-means clustering I am implementing the `k-means clustering algorithm` as described in the lectures/slides and choosing $k = 2$ as the assignment requires.

```
k = 2

# Chooses starting points for the centers
centroids = np.array([data[train_target == kk][0] for kk in range(k)])
```

```
last_centroids = centroids.copy()

# Chooses the tightness required for convergence
convergence = 1e-11

# A numpy array used to keep track of the centers each point correspond
    to
assigns = np.zeros(len(data)).astype(int)

# The main loop
while True:
    # Loops over all data points
    for i,d in enumerate(data):
        lowest = 1000
        # Find the centroid closest to the data point. The code is fast
    so I can just keep on using for-loops
        for k,c in enumerate(centroids):
            if dist(c-d) < lowest:
                lowest = dist(c-d)
                assigns[i] = k

    # Moves the centroids
    for i in range(len(centroids)):
        centroids[i] = np.mean(data[assigns == i],axis = 0)

    # If converged, return the centers
    if np.sum(centroids - last_centroids) < convergence:
        break
    else:
        last_centroids = centroids.copy()
```

Event though the code was not optimized it converged quickly. The results (i.e. a scatter-plot with the cluster-centers visualized) can be seen in figure 4.

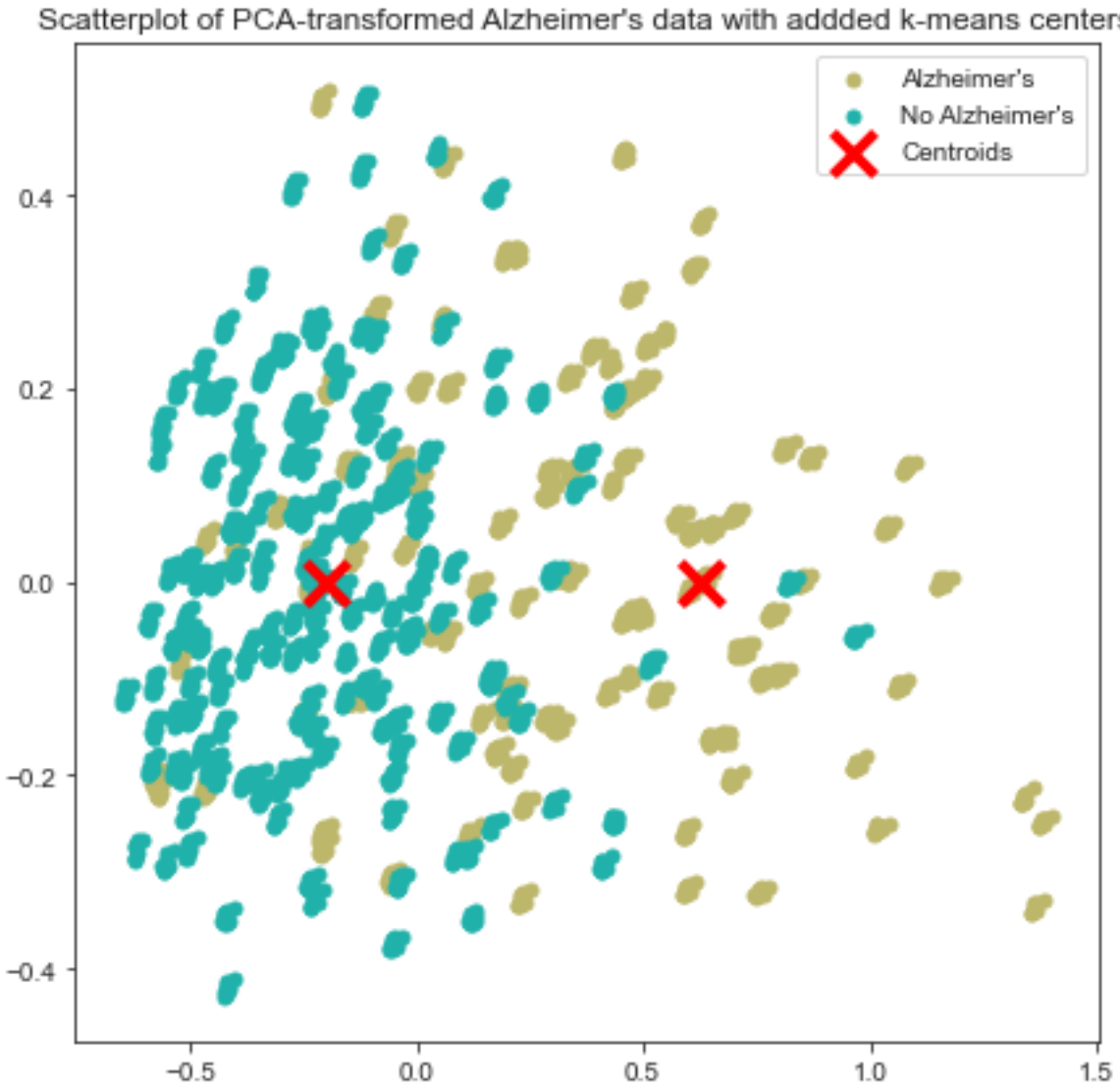

Figure 4: The same plot as figure 3 with added calculated k-means clustering centers. For the labels I have assumed that there was more data points of people without Alzheimer's than with.

1.4

1.4.1 multi-nominal logistic regression

Here I realized, that I was not asked to write my own implementations, so I used the SciKit-Learn package. Specifically I used the linear_model LogisticRegression as follows:[4]

[4]When using no regularization I had to increase the maximum number of iterations.

```
lr = LogisticRegression(multi_class = "multinomial", penalty = "none/l2",
    max_iter = 200)
lr.fit(X,y)
```

There were many penalties to choose from, so and I started out using the "l2"-regularization as this is the "squared distance"-regularization I understood the best from the course. On both the test- and training-set I got a 0-1 loss of about 0.86. Afterwards I tried without any regularization and got losses of 1.0 for both, meaning every data-point was correctly classified! This is of course preferable, as the good results on the validation set showed that there were no signs of overfitting.

1.4.2 Random forest

To set up the random forests as needed, the following parameters were used:

```
rf_sqrt = RandomForestClassifier(n_estimators = 200,oob_score = True,
   max_features = "sqrt")
rf_all = RandomForestClassifier(n_estimators = 200,oob_score = True,
   max_features = None)
```

These were then fitted on the training-set and tested on the test-set. The results can be seen in table 1.

	Square root	Full size
Out Of Bag	1.0	0.999
Train Loss	1.0	1.0
Test Loss	1.0	1.0

Table 1: The OOB-error and 0-1-loss for the random forests.

As can clearly be seen, Random Forests can be incredible out-of-the-box classifiers.

1.4.3 KNN

Again, *scikit-learn-packages* was used. I used the *KNeighborsClassifier* to do the classification and *cross_val_score* for the cross validation. Using a 5-fold validation any number of neighbors below 14 is equally good (see figure 5). This answer did not feel complete so I wrote my own double for-loop to iterate over both number of neighbors and number of cross-validation folds. The results can be seen in figure 6.

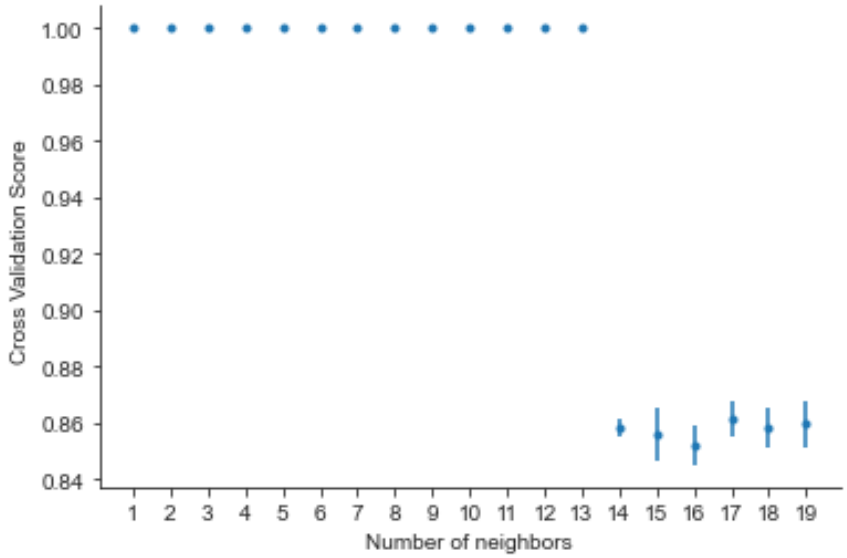

Figure 5: Using 5-fold cross validation to find the optimal number of neighbors.

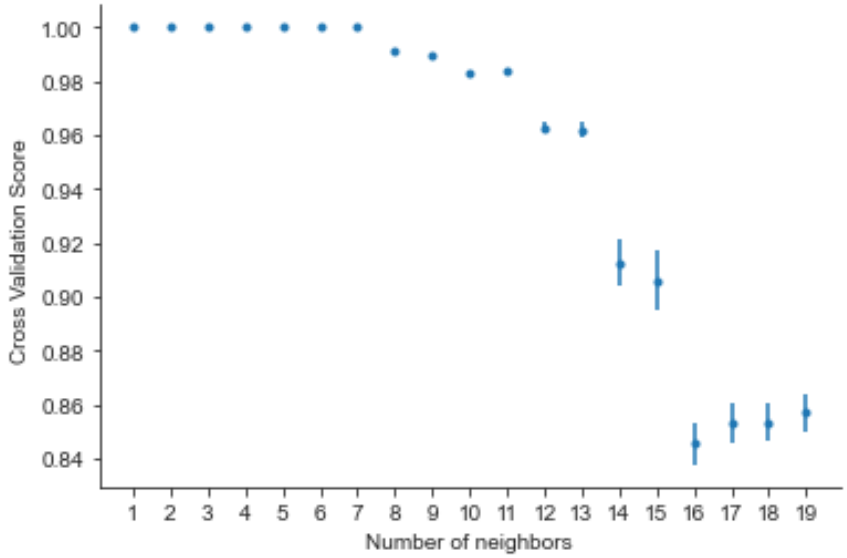

Figure 6: A mean of 2 to 20 fold cross validation to better find the optimal number of neighbors.

As can be seen in figure 6, the answer is still that any number of neighbors below 7 is good. It seemed that for our dataset, the fewer neighbors the better so I chose to go with 1. Using the training data X and their truth-labels y, the training was done by:

```
kn = KNeighborsClassifier(n_neighbors = 1)
kn.fit(X,y)
```

This gave an exact 1.0 0-1-loss for both the training-set and test-set, so for our dataset, only looking at the closest neighbor is a valid strategy!

2 Data augmentation

Deliverables: Answer the following questions briefly in your report: Which transformations are applied to the input images? Why is a transformation conditioned on the label?

```
if self.train:
    image = transforms.RandomAffine((-5,5))(image)
    image = transforms.RandomCrop((self.img_width_crop, self.
   img_height_crop))(image)
    #image = transforms.ColorJitter(0.8, contrast = 0.4)(image)
    if label in [11, 12, 13, 17, 18, 26, 30, 35]:
        image = transforms.RandomHorizontalFlip(p=0.5)(image)
    else:
        image = transforms.CenterCrop((self.img_width_crop, self.
   img_height_crop))(image)

    # The added transformation
    image = transforms.RandomPerspective(distortion_scale=0.3, p=0.5)(
   image)
```

During the training a number of built-in image-transformations are run. Looking at the official documentation[5] for torchvision it can be surmised how and why they are used:

- **RandomAffine**: Makes a random rotation of the image, padding the newly created difference with black. The (-5,5)-parameter corresponds to a maximal rotation of 5 degrees in both directions. If given a second parameter, this function could also randomly translate the image.
- **RandomCrop**: Crops the image in a random place. In the code example this is to a 28x28 square as defined earlier in the class.
- For certain images **RandomHorizontalFlip** is used. Inspecting the classes shows, that this only happens on horizontally symmetric signs (an example of which can be seen in figure 7), effectively doubling the amount of data for these classes. In the code, there is a 50% chance of flipping the image.
- If the image is not horizontally symmetric **CenterCrop** is used to crop the image to a 28x28 square in the center.

I have also added the **RandomPerspective**-function. Given that this is an computer vision problem I image this could be very rewarding for our model when testing in real

[5] https://pytorch.org/vision/stable/transforms.html

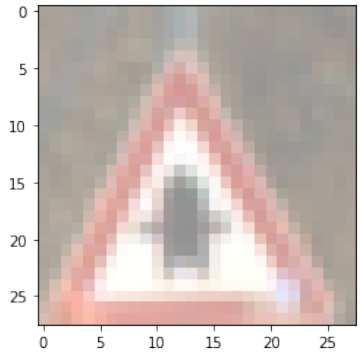

Figure 7: A horizontally symmetric traffic sign. In the data this has label 11.

life. I chose for it to only happen with a 50% chance on the images. I also set the distortion scale parameter to a semi-low 0.3, since cars will (hopefully) see the traffic signs relatively head-on. I also thought about using the **GaussianBlur** as lenses tend to get foggy, but the images are already relatively downscaled so this might not be very rewarding.

3 Efficient use of data

Starting out by using **Corollary 3.4** from the Yevgeny Seldin Machine Learning Lecture Notes we can build two probability bounds:

$$\mathbb{P}_0 = \mathbb{P}\left(\exists h \in \mathcal{H} : L(h_{1,i}) \geq \hat{L}(h_{1,i}, S_0) + \sqrt{\frac{\ln(M/\delta)}{n}}\right) \leq \delta$$

and

$$\mathbb{P}_1 = \mathbb{P}\left(\exists h \in \mathcal{H} : L(h_{0,i}) \geq \hat{L}(h_{0,i}, S_1) + \sqrt{\frac{\ln(M/\delta)}{n}}\right) \leq \delta$$

Where we assume an even split, so the $2n/2$ in the denominator is simplified to n.[6]

Now, we want to know probability for both bounds to be valid. We do this by taking the union:

$$\mathbb{P}_0 \text{ and } \mathbb{P}_1 = \mathbb{P}\left(\bigcup_{j=\{0,1\}} \exists h \in \mathcal{H} : L(h_{j,i}) \geq \hat{L}(h, S_{1-j}) + \sqrt{\frac{\ln(M/\delta)}{n}}\right) \tag{1}$$

We can now take advantage of Boole's Inequality:[7]

[6]This is bad notation, I know. I just want to avoid writing n' for the rest of the question.

[7]The Union Bound, among friends.

$$\mathbb{P}\left(\bigcup_{j=\{0,1\}} \exists h \in \mathcal{H} : L(h_{j,i}) \geq \hat{L}(h_{j,i}, S_{1-j}) + \sqrt{\frac{\ln(M/\delta)}{n}}\right) \leq \sum_{j=\{0,1\}} \mathbb{P}_j \leq 2\delta \tag{2}$$

Using a trick from one of the lectures, I negate the whole expression changing it into:

$$\mathbb{P}\left(\bigcup_{j=\{0,1\}} \forall h \in \mathcal{H} : L(h_{j,i}) \leq \hat{L}(h_{j,i}, S_{1-j}) + \sqrt{\frac{\ln(2M/\delta)}{n}}\right) \geq 1 - \delta \tag{3}$$

As the expression is now valid for all h we can find the optimal hypothesis by $h_{j^*,i^*} = \arg\min_{j\in\{0,1\},i\in\{1,\ldots,M\}} \hat{L}\left(h_{j,i}, S_{1-j}\right)$.

$$\mathbb{P}\left(L(h_{j^*,i^*}) \leq \hat{L}(h_{j^*,i^*}, S) + \sqrt{\frac{\ln(2M/\delta')}{n}}\right) \geq 1 - \delta' \tag{4}$$

This means, I have found a high probability bound. I now know that with probability at least $1 - \delta'$:

$$L(h) \leq \hat{L}(h^*, S) + \sqrt{\frac{\ln(2M/\delta')}{n}} \tag{5}$$

4 Learning by discretization

1. Derive a generalization bound for learning with H.

Even though it is not explicitly stated, I assume a loss bound by (0,1). This problem closely resemble one of the assignments, so the the Occam's Razor Bound seems like the obvious starting point. By the confidence δ, the true loss $L(h)$, the empirical loss $\hat{L}(h, S)$ and sample size n, the general formula is:

$$\mathbb{P}\left(\exists h \in \mathcal{H} : L(h) \geq \hat{L}(h, S) + \sqrt{\frac{\ln\left(\frac{1}{\pi(h)\delta}\right)}{2n}}\right) \leq \delta \tag{6}$$

(Yevgeny Seldin Machine Learning Lecture Notes, **Theorem 3.3**)
Here, $\pi(h)$ is an arbitrary function with the constraint that $\sum_{h\in\mathcal{H}} \pi(h) \leq 1$. Splitting up the short-hand notation, the constraint for our specific problem is:

$$\sum_{d\in\{1,2,\ldots\}} \sum_{h\in d(h)} \pi(h) \leq 1 \tag{7}$$

Looking at the problem, it can be seen that for each dimension d there are d^2 binary choices, giving 2^{d^2} hypotheses total. This means, choosing $\pi(h) = (2^{d^2})^{-1}$ will make the 'inner' sum

equal to 1 for each dimension. To suppress the outer infinite sum, we do the same trick as is done in the proof for **Theorem 3.5**, adding a term that follows the Geometric Series. This gives:

$$\pi(h) = \frac{1}{2^d}\frac{1}{2^{d^2}} = \frac{1}{2^{d(d+1)}} \tag{8}$$

The final:

$$\mathbb{P}\left(\exists h \in \mathcal{H} : L(h) \geq \hat{L}(h,S) + \sqrt{\frac{\ln\left(\frac{2^{d(d+1)}}{\delta}\right)}{2n}}\right) \leq \delta \tag{9}$$

Meaning that with a probability of at least $1-\delta$

$$L(h) \leq \hat{L}(h,S) + \sqrt{\frac{d(d+1)\ln 2 - \ln\delta}{2n}} \tag{10}$$

2. Explain how to use the bound to select a prediction rule $h \in H$.

Whenever a higher resolution/dimensionality d is chosen, $\hat{L}$ can be lowered, but the square-root term penalizes this, by raising the bound between the empirical and the expected loss. This is best explained visually (see figure 8), but the gist is, that for a given certainty δ we choose a grid-resolution which allows us a good empirical loss but stay aware of the bound-term that increases and 'obfuscates' the true loss by a certain probability.

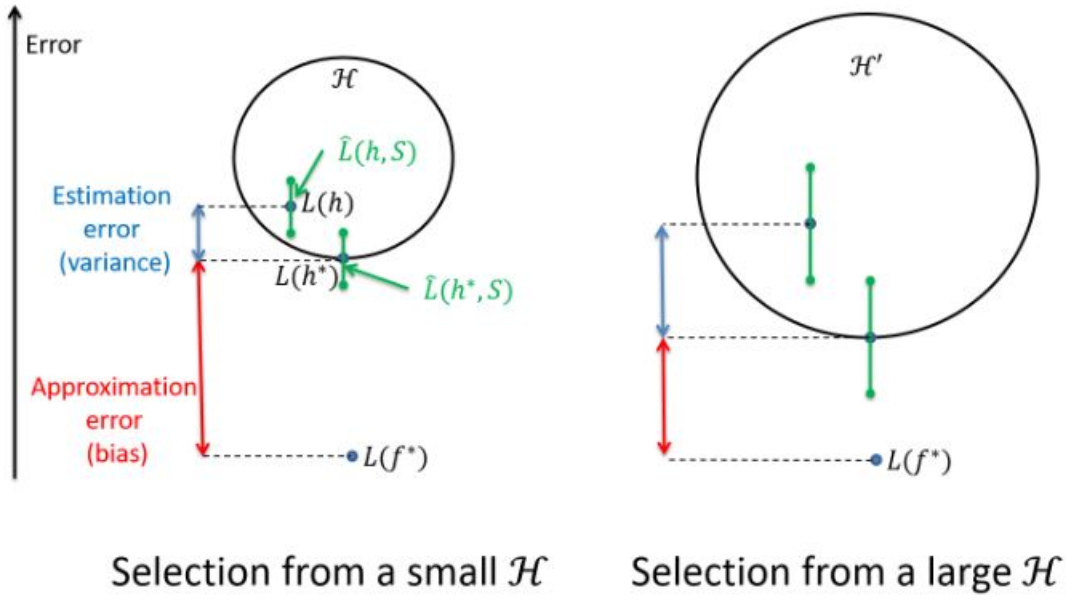

Figure 8: A visualization of the trade off between Estimation Error and Approximation Error with larger hypothesis spaces. (Figure Source: Fig. 3.2 from the Yevgeny Seldin Machine Learning Lecture Notes)

3. What is the maximal number of cells as a function of n, for which your bound is non-vacuous? (It is sufficient to give an order of magnitude, you do not need to make the precise calculation.)

As we assume a loss bounded in $[0, 1]$, any term above 1 is vacuous and trivial. Doing some algebra, this translates to:

$$\sqrt{\frac{d(d+1)\ln 2 - \ln \delta}{2n}} = 1 \leftrightarrow \frac{d(d+1)\ln 2 - \ln \delta}{2n} = 1$$
$$\rightarrow d(d+1) = \frac{2n + \ln \delta}{\ln 2} \leftrightarrow d = \sqrt{1/4 + \frac{2n + \ln \delta}{\ln 2}} - 1/2$$

This means the maximal number of squares as a function of n is the square of this function. As as I assume the dimensionality is constrained to be integers, the corresponds to:

$$\text{cells} = d^2 = \left\lfloor \sqrt{1/4 + \frac{2n + \ln \delta}{\ln 2}} - 1/2 \right\rfloor^2 \tag{11}$$

An example: For a certainty of $\delta = 0.8$ and sample size $n = 100$, d is just above 16. This corresponds to a maximum number of cells $16^2 = 256$

A good observation is, that choosing a specific δ and sufficiently high n, makes d grow about as $\sqrt{n}$ i.e. the the number of grid cells grows approximately proportionally to n.

4. Explain how the density of the grid affects the bound. Which terms in the bound increase as the density of the grid increases and which terms in the bound decrease as the density of the grid increases?

As already mentioned in (2.) the $\sqrt{n}$ term goes up, but the increased fidelity allows for a better h^* to be found either making $\hat{L}(h, S)$ fall or stay still. Again, figure 8 is the best way of visualizing what happens.

5 Regression

5.1 Decision Tree Regression

In this assignment we had to do a 5-fold cross validation to find the optimal depth of a trained decision tree. Most of these actions could be by use of SciKit-Learn, namely the *DecisionTreeRegressor* and *cross_val_score* packages. The search for the optimal max-depth was then done by the following code:

```
scores, stds = [], []
for i in range(1,16):
    dt = DecisionTreeRegressor(max_depth = i)
    score = cross_val_score(dt, X, y, cv=5)
    scores.append(score.mean())
    stds.append(score.std())
```

An example of a run can be seen in figure 9. The convergence was somewhat dependent on the random seed, but after running my script for a couple of times, it seemed that the higher the better, meaning 15 would be the optimal depth.

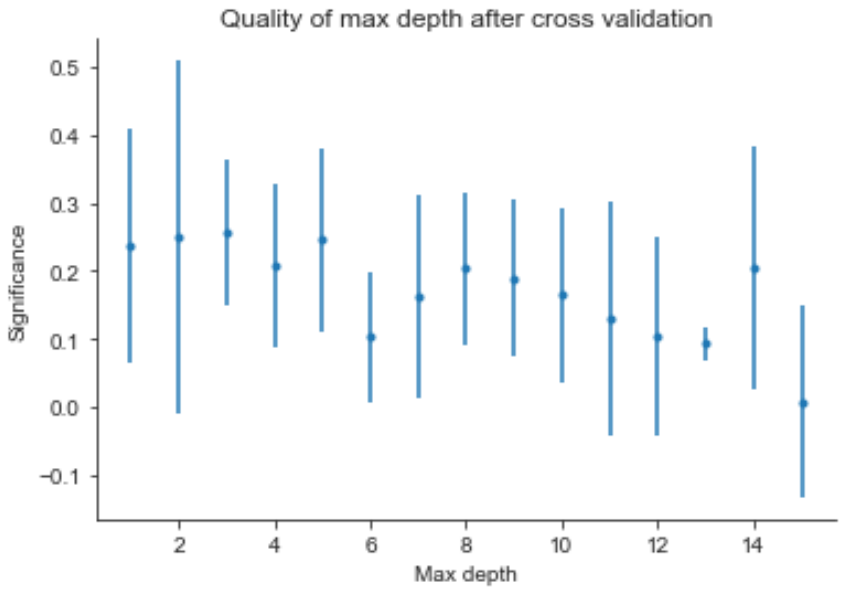

Figure 9: An example of a the results from a 5-fold cross validation trying out different max depths for a single Decision Tree on the training data set.

The tree was then trained using this depth. The quality of results from the trees can be seen in figures 10 and 11 respectively.

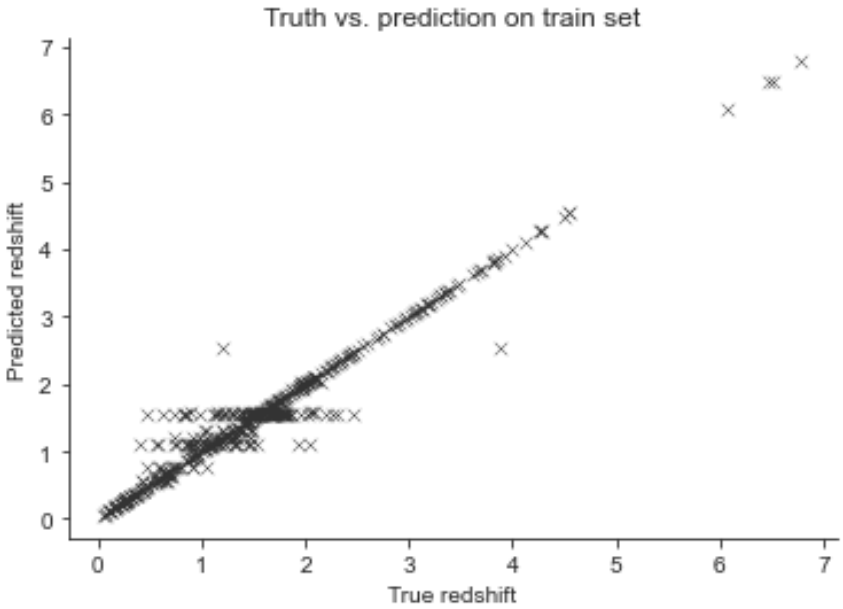

Figure 10: The results of regressions on the training set using a Decision Tree with a depth of 15. Points closer to the diagonal are better predictions.

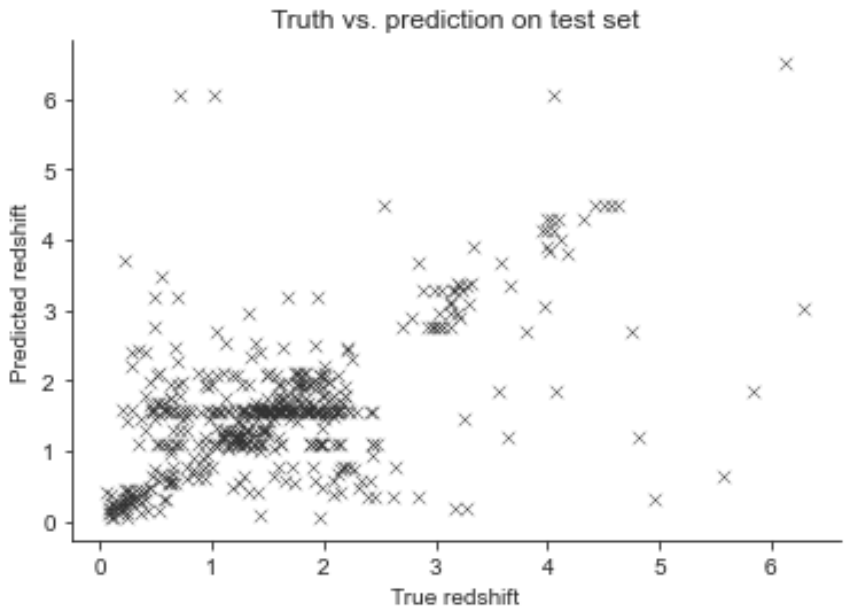

Figure 11: The results of regressions on the validation set using a Decision Tree with a depth of 15. Points closer to the diagonal are better predictions.

As is clear, there are horizontal 'lines' of misclassified redshifts. This comes from the fact that the regression model is forcibly cut off in depth, corresponding to a lowered resolution in quality, bunching multiple inputs into the same guess.
The Mean Squared Error on the training set was 0.0414 and 0.883 on the test set.

5.2 Extremely Randomized Tree Ensemble:

Firstly a random forest had to be trained. The problem was designed to require the standard parameters, so the needed code was:

```
er = ExtraTreesRegressor(n_estimators = 500)
er.fit(X,y)
```

Hereafter the Extremely Randomized Trees were made to predict on both the training-set and the test-set. The results are visualized in the following scatter plots:

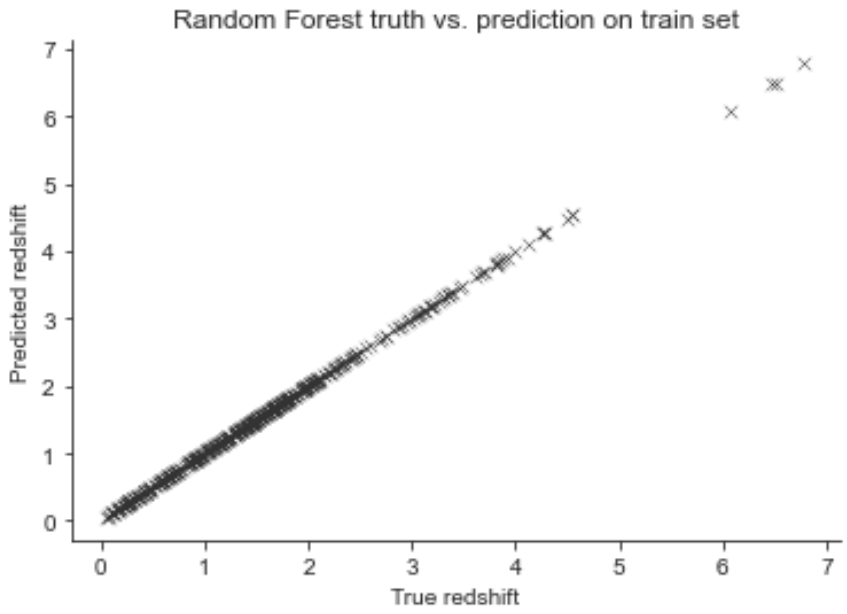

Figure 12: The results of regressions on the training set using a Forest of Extremely Randomized Trees. Points closer to the diagonal are better predictions.

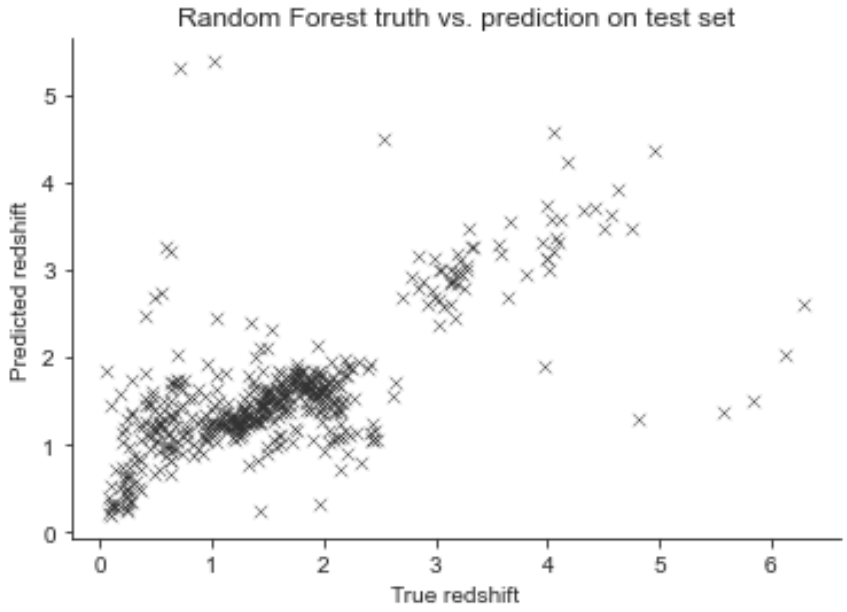

Figure 13: The results of regressions on the training set using a Forest of Extremely Randomized Trees. Points closer to the diagonal are better predictions.

As can be seen visually, both Random Forest-regressions are improvements over using a single tree. This improvement also shows, as the Mean Squared Error on the train- and

test-sets were $1.59 \cdot 10^{-28}$ (so basically a perfect regression) and 0.597 respectively.

Given an infinitely deep tree any function can be approximated,[8] so as we now allow for the trees to grow and combine 500 of them, there is no unnatural 'blockiness' in the predictions on the scale of our inputs and outputs. Much like additional pixels on a screen, the extra leafs allow for a greater quality in the regression.

5.3 Impurity

Question: Can it happen that a child of a node exhibits a worse impurity Q than the node itself?

Answer: Yes! A forest of Extremely Randomized Trees doesn't care. Every node is split on random among a random subset of the features. Any individual tree/node does not care about its individual impurity. As you asked for a toy example, you could think of a binary problem: If a node has a cut that splits the data into 90:10 between the two classes, its child node can choose to make this into 10:10. If the first class is the correct class this is an obvious downgrade in purity.

[8] See The Universal Approximation Theorem