Assignment 6

XGBoost (25 points)

1.

We load the dataset and split it into a training set containing 80% of examples, and a test set containing 20% of examples using the `train_test_split` function from `sklearn.model_selection`. A code snippet is provided below,

```
import numpy as np
from sklearn.model_selection import train_test_split
data = np.loadtxt("quasars.csv", delimiter = ",") # load data
y = data[:,10] # Labels
X = data[:, 0:10] # Features
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=120)
```

2.

We split the training data, keeping 90% of examples and using 10% as a hold-out validation set using the same procedure as above. For the XGBoost regression implementation, we use the `xgboost` library as described in the assignment. We use the parameter values given in the assignment. Per the documentation linked in the assignment, the default objective for `XGBRegressor` to optimize, is the squared loss. We monitor the training progress on the validation set, by using the `fit` method from the `XGBRegressor` class and specifying `eval_set` to be a list containing the validation set and the training set. This gives us the RMSE on both sets. A code snippet is provided below

```
import xgboost as xgb
X_train_1, X_eval_1, y_train_1, y_eval_1 = train_test_split(X_train, y_train,
test_size=0.1, random_state=120) # split into training and validation set
# Initialise XGBRegressor, with specified parameters
reg = xgb.XGBRegressor(colsample_bytree=0.5, learning_rate=0.1, max_depth=4,
reg_lambda=1, n_estimators=500)
# Fit XGBRegressor to training set
# and evaluate validation error on validation set
reg.fit(X_train_1, y_train_1, eval_set=[(X_eval_1, y_eval_1), (X_train_1, y_train_1)])
```

Below we plot RMSE on the validation and training set vs boosting rounds,

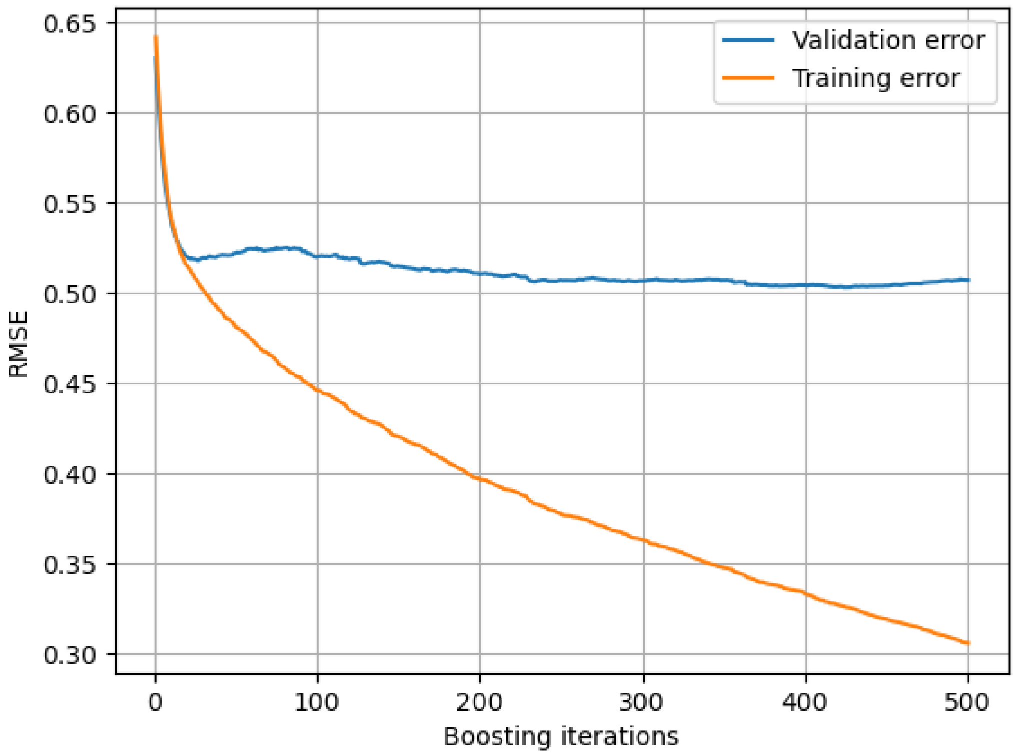

Figure 1: RMSE on train and validation set vs boosting iteration

Finally, on the test data we compute the RMSE by using the `score` method from the `XGBRegressor` class. We get that,

$$RMSE = 0.5192$$

We can compute the R^2-value by $\frac{RMSE^2}{\sum_{i=1}(y_i - \bar{y})^2}/n$, where $\bar{y}$ is the mean of the test data. We get that,

$$R^2 = 0.352406$$

3.

We do grid search to find good parameter estimates for the XGBoost model. We use the following parameter grid

- `colsample_bytree`: For the proportions of features to be checked for each tree we take the symmetric grid around 0.5, given by $\{.3, .4, .5, .6, .7\}$.
- `learning_rate`: For the learning rate we take the grid $\{0.05, 0.1, 0.2, 0.3\}$. Note that we try the, according to Fabian Gisekes slides, default value of 0.3.
- `max_depth`: For the max depth of trees we take a symmetric grid around 4. That is we try $\{2, 3, 4, 5, 6\}$.
- `reg_lambda`: For the regularisation parameter λ, we take a logarithmic grid around 1, given by $\{0.01, 0.1, 1, 10, 100\}$.

- `n_estimators`: For the number of boosting rounds, we try $\{300, 400, 500\}$. We only try values smaller than 500, due to figure 1, where we see the validation error flattening around 300-400. This could of course be because of the other hyperparameters, but we justify this by noting that the validation error is already pretty low around round 30, which is a factor 10 less than 300. In addition to this, we have a pretty large grid already, and to keep training time from exploding, it is best to consider smaller values for boosting rounds.

We do the grid search by using `GridSearchCV` class from `sklearn.model_seletion`. We specify the above parametergrid as the `param_grid` parameter, and set the `cv=3` to do 3-fold cross-validation. We retrain the model on all training examples. Finally we obtain

$$RMSE = 0.5038327$$

And

$$R^2 = 0.3902499$$

First, we notice that we have only gained a small amount of performance compared to the parameters chosen in 2. We compare to a simple 5-nearest neighbors regression. We use the `KNeighborsRegressor` class from `sklearn.neighbors` for the implementation. From this we get the following errors

$$RMSE = 0.49341$$

$$R^2 = 0.41521$$

We notice that we actually have slightly better performance of the 5-nearest neighbors estimator, and we have thus not beaten this baseline by using XGBoost tuned via cross-validation.

Majority Vote (25 points)

Throughtout, let MV denote a uniformly weighted majority vote.

1.

Consider $\mathcal{H} = \{h_1, h_2, h_3\}$ and $\mathcal{X} = \{x_1, x_2, x_3\}$ arranged in the table below, where 1 denotes a correct prediction of h_i on x_j, and 0 denotes an incorrect prediction,

	x_1	x_2	x_3
h_1	0	1	1
h_2	1	0	1
h_3	1	1	0
MV	1	1	1

We notice that $L(h) \geq \frac{1}{3}$ for all $h \in \mathcal{H}$ and $L(\text{MV}) = 0$.

2.

Reusing the same notation as above, consider,

	x_1	x_2	x_3
h_1	1	0	0
h_2	0	1	0
h_3	0	0	1
MV	0	0	0

In this example, we have $L(\text{MV}) > L(h)$ for all $h \in \mathcal{H}$.

3.

Consider $\mathcal{X} = \{x_1, x_2\}$ and $\mathcal{H}$ such that,

	x_1	x_2
h_1	1	0
h_2	0	1
h_3	1	0
h_4	0	1
$\vdots$	$\vdots$	$\vdots$

That is h_i predicts x_j correctly if and only if $i \equiv j \mod 2$. We have that $L(h_i) = \frac{1}{2}$ for all i. Furthermore, when $|\mathcal{H}| \longrightarrow \infty$, we will have a tie between hypotheses indexed by even numbers and hypotheses indexed by odd numbers. Since we can break ties arbitrarily for the majority vote, we break ties by using the prediction of h_1 when $x_i = x_2$, and using h_2 when $x_i = x_1$. This gives us $L(\text{MV}) \longrightarrow 1 = 2\max_h \mathcal{H}$, when $|\mathcal{H}| \longrightarrow \infty$.

4.

Let $|\mathcal{H}| = M$, such that we can write $\mathcal{H} = \{h_1, h_2, \cdots h_M\}$. Assume $L(h) = \frac{1}{2} - \epsilon$ for all h in $\mathcal{H}$. From the proof of theorem 3.33, we have that,

$$L(\text{MV}_\rho) \leq \mathbb{P}(\mathbb{E}_\rho\left[\mathbb{1}(h(X \neq Y))\right] \geq 0.5) = \mathbb{P}(\mathbb{E}_\rho\left[\mathbb{1}(h(X \neq Y))\right] - L(h) \geq \epsilon)$$

We can compute the expectation by using that ρ is uniform, and that the expectation of an indicator function is the probability of the set,

$$\mathbb{E}_\rho\left[\mathbb{1}(h(X \neq Y))\right] = \sum_{i=1}^{M} \rho(h_i)\mathbb{P}(h_i(X) \neq Y) = \frac{1}{M}\sum_{i=1}^{M} L(h_i(X), Y)$$

We thus have,

$$L(\text{MV}_\rho) \leq \mathbb{P}\left(\frac{1}{M}\sum_{i=1}^{M} L(h_i(X), Y) - L(h) \geq \epsilon\right)$$

Since the losses are independent and all equal to $L(h)$ we can apply corollary 2.5, to obtain

$$L(\text{MV}_\rho) \leq \mathbb{P}\left(\frac{1}{M}\sum_{i=1}^{M} L(h_i(X), Y) - L(h) \geq \epsilon\right) \leq e^{-2M\epsilon^2}$$

Which converges to 0 when $|\mathcal{H}| = M \longrightarrow \infty$. Which is what we wanted to show.

Data Visualisation (25 points)

1.

We do PCA and t-SNE by using the `PCA` and `TSNE` classes from `sklearn.decomposition` and `sklearn.manifold`, respectively. For the PCA we disable whitening by specifying `whiten = False`. For doing t-SNE we choose the `init` parameter PCA for stability. We stick to the default parameters for the rest of `TSNE` - notably a learning rate of 200 (for the version of sklearn I am using), and a perplexity of 30, which is consistent with Sadeghs slides, suggesting choosing perplexity between 5 and 50. Code snippets are provided below

```
from sklearn.decomposition import PCA # Import PCA
pca = PCA(whiten=False, svd_solver = "full") #Intialize PCA
pca.fit(data) #Fit PCA to data

from sklearn.manifold import TSNE # Import TSNE
tsne = TSNE(init="pca") # Initialize TSNE
```

We then transform the our input data. For PCA, we project the data onto the two first principal components, by extracting the principal components from the `components_` attribute of the `PCA` class, and do matrix multiplication of these with the data-matrix. For transforming the data with *t*-SNE, we utilize the `fit_transform` method from the `TSNE` class. We provide code snippets below

```
two_principal = pca.components_[0:2] # Extract principal components
train_projection = two_principal @ data.T
# Calculate the projection of data
# onto first two principal components by matrix multiplication
trans = tsne.fit_transform(data) # fit TSNE to data
```

2.

We plot the data in 3-dimensional space before transformation below,

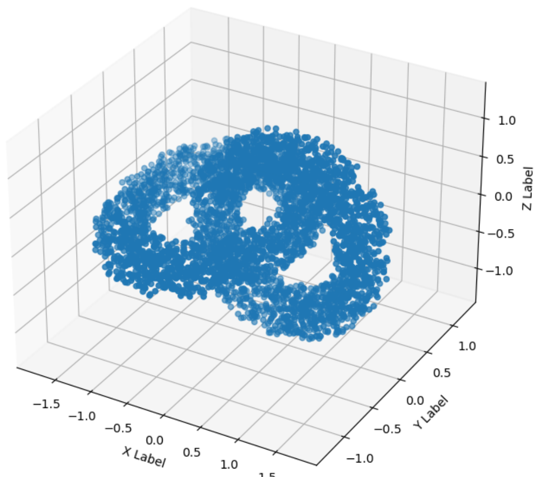

Figure 2: Non-transformed data

We see that this looks like two intertwined toruses slightly angled. We plot the data transformed by PCA,

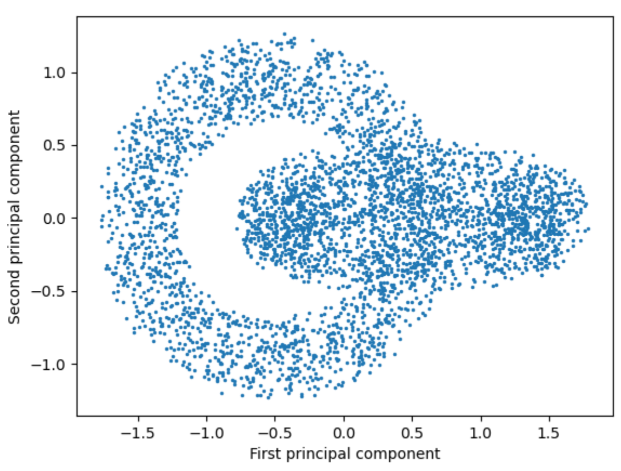

Figure 3: PCA transformed data

And finally the *t*-SNE transformed data,

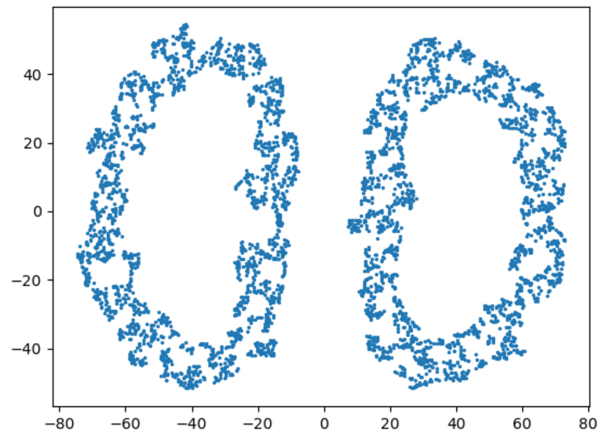

Figure 4: *t*-SNE transformed data

3.

The PCA transformed data looks like a 'flattened' version of the two toruses, whereas the *t*-SNE transformed data has two disjoint ellipsoids likely corresponding to the two toruses. This is because PCA tries to capture the variance of the data, and then transforms it linearly, meaning that the 3-dimensional shapes are directly 'flattened' when projected into 2-dimensional, wheras *t*-SNE allows for a more 'abstract' sense of closeness, such that the toruses can be more easily seperated.