Project 3 FYS-STK3155

Abstract ¶

In this project we have demonstarted the power of ensamble methods. And money. With comparing random forests, bagged, and boosted trees to linear regression as well as a feed forward network, we have shown, using United Nations happiness index, that money can buy happiness. Our results poits towards ensamble methods being among the most powerful machine learning algortihms when doing regression analysis.

1.1 Introduction

In this project we wish to find a model able to predict the happiness in a country. We use the World Happiness Report (United Nations, 2019) , detailing the happiness score of 156 countries or regions, along with features such as GDP pr. capita, social support, etc. All of the features in the dataset will be outlined at a later point.

The First step of our investigation will be principal component analysis (PCA). PCA is - simply put - a way of investigating how much each feature contributes to the total variance of the dataset. By doing this we can possibly reduce the size of our dataset by eliminating the principal components (PC) with the least contribution to the total variance, and by doing this reduce the computation time with little increase in error.

When the PCA is conducted we can begin modelling the data, and for this we will use a variety of methods. We will initially attempt to use linear regression, with and without regularization: L1 and L2 regularization, respectivly. These mothods will firstly be compared to a feed forward neural network (FFNN) developed (as in written, not conjured), and implemented in (Hagen, 2020a).

Thereafter we will use descision trees to do further modelling. We will experiment with a single tree, as well as several tress (an ensamble), and use such techniques as bagging and boosting to improve upon these ensambles.

2.1. Principal component analysis

Let's assume we have a feature matrix $X \in \mathbb{R}^{n\times p}$. This is a matrix where each coulumn represents a feature, and each row represents an observation. Now let's introduce a matrix $B = X^T - \bar{X}^T$, that is, the centered transpose of X. Calculating the covariance matrix C of B yeilds

$$C = B^T B \in \mathbb{R}^{n\times n}$$

Then, by computing the eigenvectors and eigenvalues of C we get

$$V = [v_1, v_2, \cdots, v_k]$$
$$D = [\lambda_1, \lambda_2, \cdots, \lambda_k]$$

where the indices rank the eigenvalues by magnitude; the lower the index, the higher the value, or:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$$

We can now write the k number of eigenvalue equations as $CV = VD$.

We can now define $T = BV$, where T is the principal components (Hjorth-Jensen 2020).

Alternatively we can use singular value decomposition (SVD) to define

$$B = U\Sigma V^T \rightarrow T = U\Sigma$$

Since a given eigenvalue $\lambda_p = \sigma_p^2$, that is: the variance of the data of feature p, we can find the proportion of the variance contributed by each eigenvalue, and, by extention, each feature:

$$P_i = \frac{\lambda_i}{\sum_{j=1}^{k} \lambda_j}$$

where P_i is the proportion of variance contributed by the i-th feature

Applying this analysis to our dataset we can construct a new feature matrix, only consisting of the the features most contributing to the variance and hopefully use less time running our algorithms.

2.2 Linear regression and gradient descent

The first part of analysis will be consisting of methods we have utilized several times previously. Namely linear regression (LR) (Hagen, 2020b), and a feed forward neural network (Hagen, 2020a).

In the case of LR, we will use both L1 and L2 (Lasso and Ridge, respectively) regularization with cross validation.

As discussed in (Hagen, 2020b) LR, and in particular, ordinairy least squares, is defined as

$$y = X\beta + \epsilon$$

where X is the feature matrix, and ϵ is the noise in the data. β is the solution to

$$\beta = \text{argmin} MSE = \text{argmin} C(\beta) = \text{argmin} \frac{1}{n} \sum_{i=1}^{n} (y - X\beta)^2$$

whose solution is $\beta = (X^T X)^{-1} X^T y$, where y is the target data.

A ploblem my arise where $X^T X$ is a singular matrix, thus not being invertible. If this is the case we could find the pseudo inverse (Wikipedia 2020) of $X^T X$ by way of SVD (singular value decomposition). An alternative trick is to introduce a regularization parameter, in order to dispense with the singularity of $X^T X$:

We will the let

$$\beta = (X^T X + \alpha I) X^T y$$

in other words: we add a parameter α to the diagonal of the singular matrix, making it non-singular. This is what is reffered to as L2 regularization, yielding the optimization problem:

$$\min_{\beta}(y - X\beta)^T(y - X\beta) + \lambda(\beta^T\beta - c) \tag{1}$$

An alternative scheme is to introduce the parameter β_0 in the cost function, such that

$$C(\beta) = \sum_{i=1}^{1}(y - \beta_0 - x_i^T\beta)^2 \tag{2}$$

, with the constraint

$$\sum_{j=1}^{p}\beta_j \leq t$$

where t is an adjustable parameter

Unfortunatly, we cannot analytically solve for the minimum of (1). This can be done numerically by, for example, mini-batch gradient descent, an algorithm outlined in (Hagen, 2020a). The algorithm is presented in (3)

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{3}$$

The the basic principal of mini-batch GD is to split the data up into B_k minibatches. Then compute the gradient for each bacth and update β. This method is particularly useful not only when the minimum of the cost function cannot be found exactly, but also when there are several minima to be found. Obviously we want to find the global minimum, but it's not a given that this is what we find initially. So by constantly updating β with the gradient of different batches we can escape these local minima and hopefully find, or at least approach, the global minimum.

The γ in (3) is called the learning rate. The reason we have a learning rate might not be self evident, so I'll attempt to put it as simply as possible. For the sake of simplicity we will concider a parabolic cost function, as presented in figure 0.1

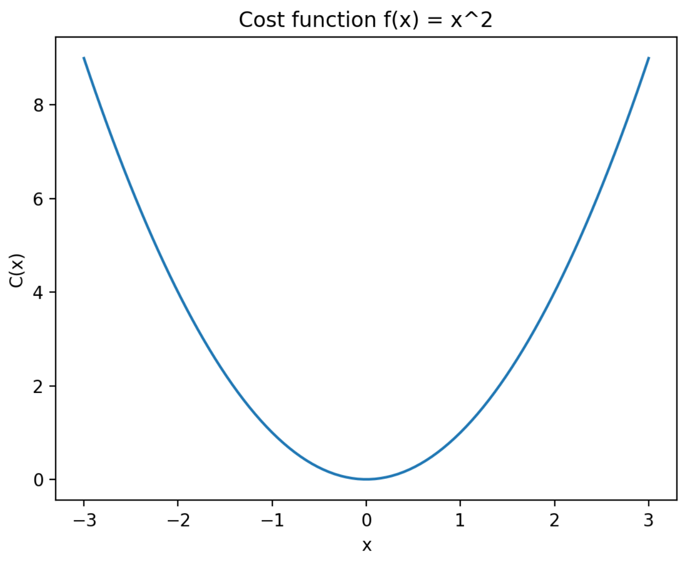


Figure 0.1: Hypothetical cost function

We know that the global, and only, minimum of this function is at $x = 0$. But let's assume for a second that this information is not available to us. What are we to do then? With gradient descent we make a guess as to where the minimum is. From the graph we can see that it's somewhere around $x = 0.5$. So we can that say that $x_0 = 0.5$, and then use gradient descent to approach the true minimum.

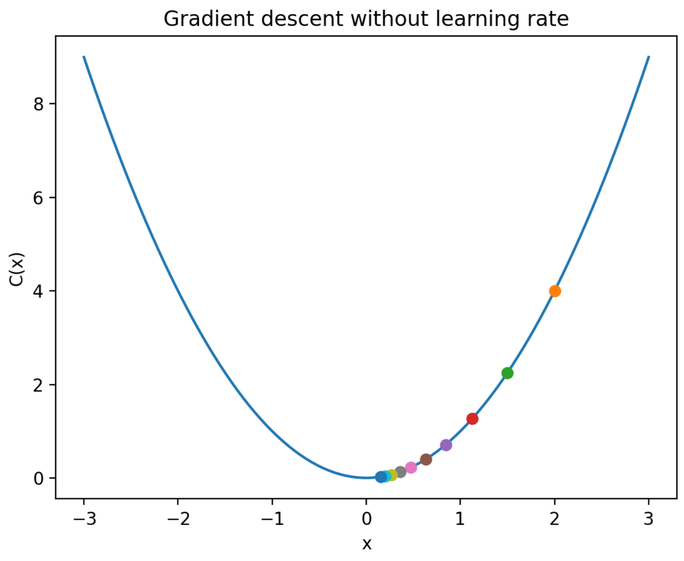


Figure 0.2: Gradient descent on costfunction without learning rate

We can see from figure 0.2 that it takes quite a few steps to reach the minimum. To remedy this we scale the gradient by the factor γ so that it takes bigger steps towards the minimum than it would without. At the same time, we decrease the learning rate incrementally so that we don't overshoot, and end up further away from the minimum, or bouncing around it. In figure 0.3 I've illustrated the result.

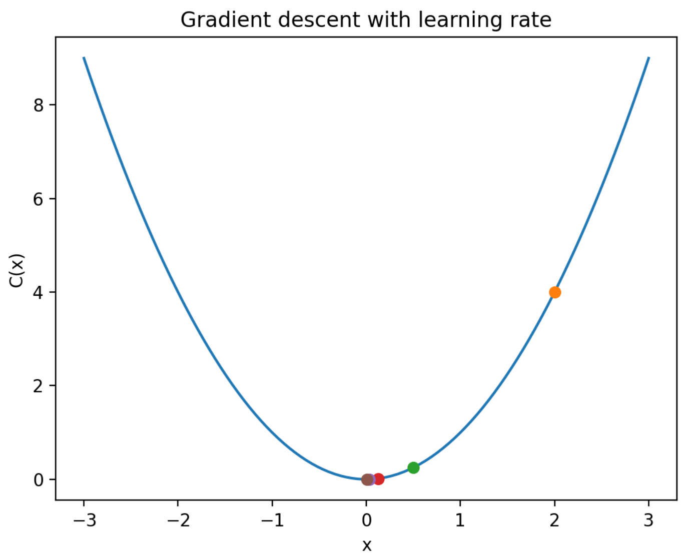


Figure 0.3: Gradient descent on cost function with diminishing learning rate

In other words: we get at the minimum at a much quicker rate than we otherwise would have done without the learning rate.

This is the basic principle of gradient descent, and why we use the learning rate γ

So to reiterate:
We split the data into B_k batches
For each batch we compute the gradient and update β
This defines one epoch.

We can do this for a given number of epoch, shuffeling the dataset each time in order to avoid computing the same k gradients over and over.

When this algorithm is run we are left with what is hopefully a good estimate of the global minimum of the cost function.

2.3 Feed forward neural networks

Before you read this part: This explanation has already been done in (Hagen 2020a), but (according to the author) to an unsatisfactory degree. This section is an attempt to, to some extent, rectify ambiguities, unclarities, etc from earlier. Feel free to skip this part (and move on to 2.4) if this section is of no interest to you.

The assumtion we have made so far is that the coefficients of data is linear. This might not always be the case.

In the case of linear data the best approach is to use LR, not only from the fact that it is in this case a fair/ correct assumtion, but it's quick and easy.

In the case of non-linear data we need a different approach. Which? That's not always an easy question to answer. We have a myrriad of different methods we can implement, which can (just about) be sorted into two different categories. Blackboxes, and whatever the opposite of a blackbox is.

A blackbox model is a model where you essentially are blind to how the model works. You can train it on some input data, feed it some test data, and you get a prediction. The process at which blackbox methods arrive at their answer are in a sense invisible to us. We can perhaps see the architecture, but the model by which they output their prediction is not something we can easily extract or use directly.

Neural networks fall into the blackbox category, and is incidentally what we initially will use as a comparison to the linear regression methods.

We will now, more spesifically, concider what is called a feed forward neural network. A feed forward neural network (ffnn) is perhaps the simplest of neural networks.

A ffnn is an "architecture" consisting of several layers. A layer is a series of neurons/ nodes, each with a certain bias associated to it. There are three different "types" of layers in the ffnn we will be discussing here. There is an input layer, an output layer, and there are hidden layers. The input layer recieves the data we want to train our model on. The input layer must have the same number of nodes as our data has features. Each node in the input layer is connected to every other node in the proceeding layer: the first hidden layer. Each individual node-to-node connection has an associated weight. This is true for every individual node in the entire network. The inputlayer then proceeds by feeding the input data into the first hidden layer. When the data is sent on to the next layer it gets multiplied by the weights between the node-to-node connection, and the bias of the following node is added.

One of the key aspect of a neural network is that it's task is not make a model per se, but rather to emulate how neurons work in the brain. So how do they work? Simply put, neurons activate when their input exceeds a certains threshold. So you could say that each neuron in the brain has a certain threshold function associated to it. The purpose of this threshold is to ensure that a neuron doesn't fire at random causing an epileptic seizure, but rather only when it's "called upon" to fire. We wish to implement this functionality in our neural network. This threhold function is in the context of a neural network reffered to as an activation function.

So: when a node in the first hidden layer recives an input (weighted with a bias added), we want to control whether or not the node should have an output, or at least the magnitude of the output. This is done by feeding the input of the node into an activation function (we'll get back to these shortly), and the output of the activation function will be the output of the node. This is done for every node in the first (and all consecutive) hidden layer(s).

The size of the hidden layer has not yet been discussed, and for good reason. There is no clear answer to what the optimal size is. Or how many hidden layers there should be, but I'm getting ahead of myself.

Now we can choose whether we want more hidden layers or not. If yes, then we can send the output from the first hidden layer to the nex, much like we did from the input layer to the first hidden layer. If we don't want any more hidden layers we feed the data to the output layer, in the same fashion as we would to a hidden layer.

In this project we will not be discussing the use of neural networks in classification cases, so what comes next applies exclusively to what we're actully working on: regression.

In the regression case, we don't want an activation in the output layer beacuse that is more closely related to probabilities in classification cases. You can read more about this in (Hagen, 2020a). In the regression case we use one node in the output layer, and this node will output the networks prediction $\tilde{y}$.

The $\tilde{y}$ produced by the network will then be compared compared with the target value, and we can then compute the error made by the network. Then we propagate this error backwards through the layers, to update the weights and biases. This part is more conveniently explained by the mathematics of the problem

Let's go back to the beginning. The input layer sends the input data to the first hidden layer. This can be represented as

$$z_k^1 = \sum_{j=1}^{M} w_{kj}^1 x_j + b_k^1 \tag{4}$$

Where z_k^1 is the input of the k-th node in the first layer. $w_{k,j}$ is a matrix of weights associated with the k-th node, where the index j refers to the j nodes in the previous layer. x_j is the data, and b_k is the bias of the k-th node in the layer

The nodes then output $f(z_i^1) = a$, where f is the activation function we mentioned earlier. This is done for each layer all the way to the output layer.

Now we can concider the error in the output layer. We'll let the index L denote the output layer, and the index l denote an arbitrary hidden layer. Now we assume that whatever error we got, it originates from the wieghts and biases. So we want to adjust them in order to minimize the error. We do this by first defining a cost function $C(W^L)$, whose argument is the output layer weight matrix for the output layer. Then we differentiate is with respect to the weights of each node.

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1} \tag{5}$$

Where δ_j^L is the error in the input node k of the outputlayer (L), and a_k^{L-1} is the activation of the k-th node of the previous layer

We can write

$$\delta_j^L = f'(z_j^L)\frac{\partial C}{\partial (a_j^L)} = \frac{\partial C}{\partial b_j^L} \tag{6}$$

where b^L is the bias of the nodes

By using (6) we can find the error of each previous layer by

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{7}$$

This is what is known as the back propagation algorithm, because we start at the output layer, and work ourselfs backwards, all the way to the first layer.

For each layer we update the weights and biases:

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1} \tag{8}$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l} = b_j^l - \eta \delta_j^l \tag{9}$$

Let's take a step back and evaluate what we've got so far. We've fed traing data though the layers, weighting them, adding biases, and evalueted the error. We have then propagated that error backwards through the layers, adjusting weights and biases along the way. What's next?
What we've described until now we can define as one iteration. After this iteration we repeat the process for as many iterations we deem necessary. When we've run through all these iterations we concider the neural network to be trained, and we can finally use it to make predictions.

I've earlier promised to touch upon what the activation functions actually are. We can begin by simply defining the perhaps most commonly used

$$f_{Sigmoid}(z) = \frac{1}{1+e^{-z}} \tag{10}$$

$$f_{RELU}(z) = z^+ = max(z, 0) \tag{11}$$

$$\begin{gathered} f_{Leaky}(z) = z, z > 0 \\ f_{Leaky}(z) = 0.01z, else \end{gathered} \tag{12}$$

These are called, respectivly, the Sigmoid function, RELU, and Leaky RELU

Their purpose is to scale the activation of each node to an apropriate level. The Sigmoid function ensures that the output of a node is always somewhere between 0 and 1. The RELU function lets the output of the node be equal to the input, provided the input is greater than or equal to 0. Otherwise the output is 0. Leaky RELU is the same as RELU exept that for inputs smaller than 0 the output is $0.01z$, where z is the input

Which activation function to use is not obvious. We will try all three of them, comparing run time, and error.

2.4 Decision trees

After having spent some time going through in, perhaps, excruciating detail previously discussed methods and algorithms we will now enter the realm of trees. Descition trees, or more precisely, regression trees.

In comparison to neural networks, and even linear regression (depending upon your disposition), descition trees are in their formulation and execution remarkably simple creatures. So simple in fact that (Hastie et al, p 305) was able to pretty eloquently summarize them in one sentance:
"Tree-based methods partition the feature space into a set of rectangles, and then fit a simple model (like a constant) in each one"
But I suspect a little more explenation is in order.

Let's assume we have som response (target) y, taking the inputs x_1 and x_2. We can partition the plane spanned by x_1 and x_2 into n rectangles. We call these rectangles regions, and label them R_i. Each region has corresponding response values, so we find , for example, the mean of these values for every region. We label this value c_i

The question now is: how do we define the regions? We could simply make each region a square, so that we are left with a grid, but this doesn't serve. The problem is that some regions, or some of the data if you will, are difficult to predict. Taking the mean of an arbitrarily defined region might entail making assumtions about the model we are building that are simply not true

To amend this problem we can use recursive binary partitions. This is a process in which we start by first splitting the feature space into two regions: we draw a partition line, and fit a model. At this point our model will give us the mean of the responses on of the sides of the partition line. We can check the error of this prediction, and then adjust the position of the partition line to minimize the error.

We're then left with a binary partition of the feature space that best predicts the response. From here we can do the same thing to either region we now have made, and repeat the process. We stop this process when, for example, there is only a certain number of values associated with a given region.

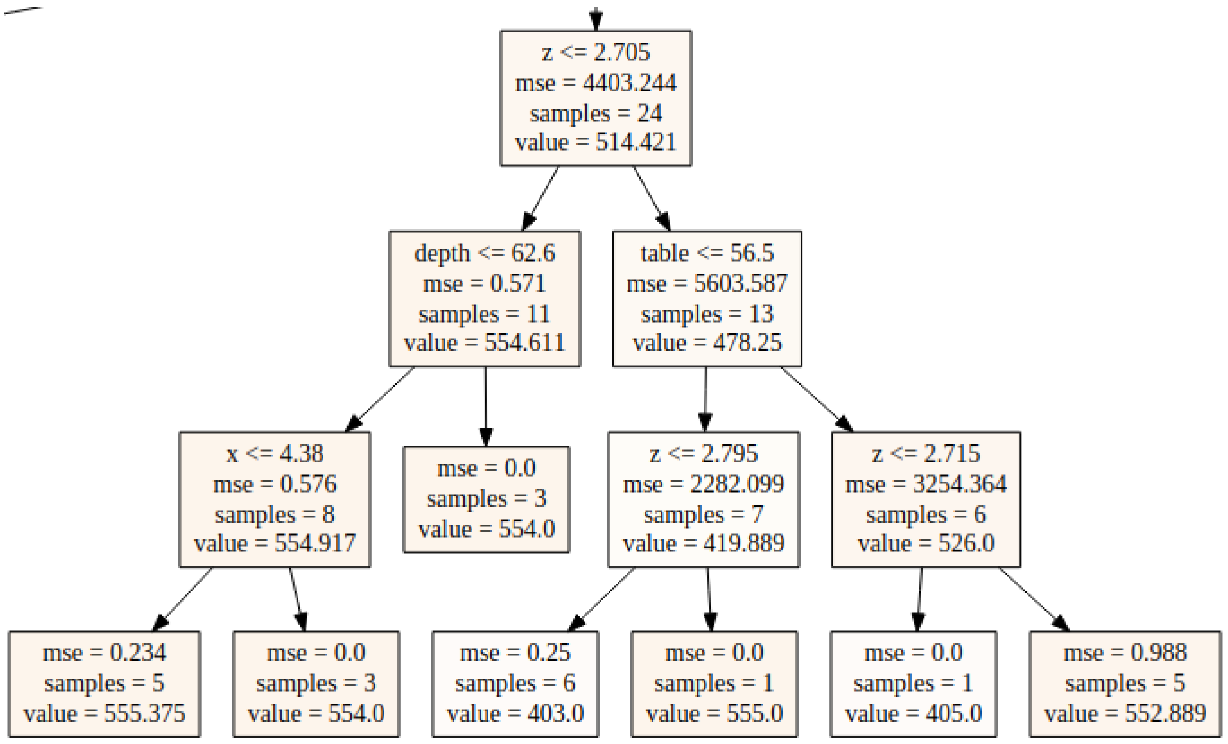
When we have partitioned our feature space into a satisfactory n number of regions R_i, each with a mean value c_i our model then becomes

$$f(x_1, x_2) = \sum_{i=1}^{n} c_i I[(x_1, x_2) \in R_i] \tag{13}$$

A visualization of the architecture of a tree might be required:

In [2]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "https://lh5.googleusercontent.com/d5QSiPN3bIKdLCWkdOlnRLpLleMMc
```

Out[2]:



Starting at the top: The first node partitions the feature space into two regions, and receives a value. Then this node asks: in which region does the value belong. Then we move on to the next node, which further paritions the region to which it belongs. Then it checks which of these regions the response belongs to. This continues until some termination criterion is imposed.

These termial nodes are often reffered to as leafes, and are usually the prime suspect when experiencing overfitting in these kinds of models. Imagine that we have a dataset, and we want to train a tree to make some prediction from this data. When we're constructing a tree we can in principal partition the feature space in such a way that each response has its own region, resulting in a perfect fit. When we then make some predicition using this trained tree, we will obviously experience gross overfitting.

The previous example was a bit of hyperbole on my part, but it gets the point across. We can quite easily overfit this model. The solution to this could be to not construct as many regions as in my example, in other words: limit the depth or the tree. Another solution is a technique called pruning. This is a method in where you remove leaves prone to overfitting. The latter is only mentioned here, as we will get back to that shortly.

So how exactly does one use a tree for regression?

Let's go back to (13), but we assume a single variable function. We model the response y with

$$f(x) = \sum_{i=1}^{n} c_i I[(x) \in R_i] \tag{14}$$

Remember when I mentioned that we draw the partition line where the model gives us the lowest error? Good. How do we find this? The sum of the square of the residuals. Finding this optimal partition line can be impractical. What we do is the following:
Let's define a splitting variable j and a split variable s, and then define

$$R_1(j,s) = \{X|X \leq s\}$$
$$R_2(j,s) = \{X|X > s\}$$

We then want to find s and j so that we can solve

$$\min_{j,s}\left(\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_r(j,s)} (y_i - c_r)^2 \right)$$

The inner minimizations are solved by

$$\hat{c_1} = avg(y_i|x_i \in R_1(j,s))$$
$$\hat{c_2} = avg(y_i|x_i \in R_2(j,s))$$

avg denoting the mean.

The outer minimization is done parsing all inputs and determine the best values for j and s. This is one of several algorithms for optimally partitioning the feature space. Other methods will yield different, and perhaps better results.

Now we can get back to pruning. Let's imagine we grown a tree, so that when a leaf reaches a certain size it becomes termial. That is: when only (for example) 20 values fulfills the criterion of the node, the node becomes terminal.
At this point we have a large tree T_0, prone to overfitting. We define a sub-tree $T \subset T_0$. We refer to terminal nodes by the index j, corresponding to region R_j. Then we define an N_j that counts the number of values contained in R_j:

$$N_j = \#\{x_i \in R_j\}$$

We can use this to find the average value in region R_j:

$$\hat{c}_j = \frac{1}{N_j} \sum_{x_i \in R_j} y_i$$

From this we can find the error S_j:

$$S_j = \frac{1}{N_j} \sum_{x_i \in R_j} (y_i - \hat{c}_j)^2$$

Resulting in the cost function

$$C_\alpha(T) = \sum_{j=1}^{|T|} N_j S_j + \alpha|T| \tag{15}$$

called the cost complexity function.

From previous experience we might be able du deduce that the point of this is to minimize this cost function. This can be done by finding a subtree $T_\alpha \subset T_0$ to minimize the costfunction, for each α. We define $\alpha \geq 0$ as a tuning parameter, dictating the depth of the tree.

Now I hope you will pardon how vague I've been in the desciption of the puning algorithm I've just detailed. The algortihm is not of much importance to this project, which is not to say that it is a useless method overall. We are more generally interested in how regression trees work, and not as much in optimizing them. We will in fact not spend much time at all modelling with a single tree.

2.5 Bagging and random forests

Many people would tell you "less is more", but this oxymoronic idea should have been dispensed with at the moment of its conception. More is more, and that is why we in this section will go from a regression tree to a forest. A random forest. But first we need might need a refresher on bootstrapping, and even introduce the concept of bootstrap aggregating (bagging).

As discussed in (Hagen, 2020b) bootstrapping is a resampling techinque used in assessing statistical accuracy. Bootstrapping is procedurally a quite simple technique:

We start by preparing a set of training data of size n
Then we draw, at random, n samples from this training data, with replacement
We fit a model to this randomly selected data.
Repeat B times
Then we assess the behaviours of all of these fits.

Now let's say that all of these fits constitue a set $f = \{f^{*_1}, f^{*_2}, \cdots, f^{*_B}\}$. The bagging estimate is the average of all of these values:

$$f_{bag} = \frac{1}{B}\sum_{b=1}^{B} f^{*_b} \tag{16}$$

You might now be able to spot why bagging is introduced. Let's say we draw a bootstrap sample from a set of training data. If we then

1) Grow a tree (like we discussed in section 2.4) from the bootstrapped data.
2) Draw a new set of bootstrapped data, and repeat 1) B times

We now have a set of trees, and to make a predicition we average the result of the predictions of all of the B number of trees. This is what we may call a bagged tree.

So far we have in large part focused on a response y only being dependent on one variable or feature x. This is slightly problematic, as the dataset we will be analyzing in this project is multidimensional. This lands us at a very critical question, and constitutes the key part of random forests: When we draw a partition line, with which feature do we do this with respect to? One answer might be to categorically work "from left to right". Let me explain.

In the data set we'll be analyzing the response is a happiness index, and the features are GDP per capita, social support, and a few others. We could build a tree by letting the first node partition GDP per capita, the second node partition social support, etc.

In random forests we do it a bit differently. When performing the bagging algorithm, in step 1): instead of growing a tree based on the the full bootstrapped dataset, we grown a random forest tree from a randomly selected subset of the features.

And there it is. A random forests is essentially a bagged tree, but the constituent trees are grown from a randomly selected subset of features.

2.6 Boosting

Boosting was originally introduced (in the realm of classification) with the idea that many "weak learners" are better than one strong learner. In the case of trees this would mean that several more shallow trees are better than one fairly deep tree. This idea has been extended to regression cases as well.

First we need a little set-up for the different boosting methods.

In the most basic sense a boosted tree is constructed thusly:
1) Grow an initial, and shallow, tree.
2) Assess the error made by this first initial tree.
3) Build a consecutive tree whose target is the error made by the previous tree.
4) Repeat 1-3 for a given number of iterations
5) Sum all the trees

Now, let's more formally define a tree:

$$T(x;\Theta) = \sum_{j=1}^{J} \gamma_j I(x \in R_j)$$

Where $\Theta = \{R_j, \gamma_j\}_1^J$. These parameters can be found by

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{x_i \in R_j} L(y_i, \gamma_j) \tag{17}$$

We can approximate (15) with

$$\tilde{\Theta} = \arg\min_{\Theta} \sum_{i=1}^{N} \tilde{L}(y_i, T(x_i, \Theta))$$

As mentioned earlier, we can represent a boosted tree as a sum of trees:

$$f_M(x) = \sum_{m=1}^{M} T(x;\Theta_m) \tag{18}$$

where we can find $\hat{\Theta}_m$ by

$$\hat{\Theta}_m = \underset{\Theta_m}{\arg\min} \sum_{i=1}^{N} L(y_i, f_{m_1} + T(x_i;\Theta_m)) \tag{19}$$

where $\Theta_m = \{R_{jm}, \gamma_{jm}\}_1^{J_m}$

Given a region R_{jm} we can find $\hat{\gamma}$ from

$$\hat{\gamma} = \underset{\gamma_{jm}}{\arg\min} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1} + \gamma_{jm})$$

Gradient boosting might be one of the simpler methods for solving these optimization problems numerically, for an arbitrary loss function L. The algortihm is layed out in the following.

1) Make an initial prediction $f_0(x) = \text{argmin}_\gamma \sum_{i=1}^N L(y_i, \gamma)$
2) For $m = 1, 2, \cdots, M$:

   a) Compute the residuals $r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i}, \quad f = f_{m-1}$

   b) Fit a tree to the residuals from a), with terminal regions $R_{jm}, \quad j = 1, 2, \cdots, J_m$

   c) For $j = 1, 2, \cdots, J_m$ compute:

$$\gamma_{jm} = \underset{\gamma}{\arg\min} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1} + \gamma)$$

   d) Update $f_m = f_{m_1} + \sum_{j=1}^{J_m} \gamma jm I(x \in R_{jm})$

3) Output $\hat{f} = f_M$

The second boosting method we'll introduce is a modified AdaBoost. AdaBoost, originally AdaBoost.M1, was developed to work on classification problems. It has later been modified in severeal different ways, one of which is AdaBoost.R2. This is the "version" we will use, as this is a boosting method we can use in regression. The full derivation of this algorithm can be found at (Drucker, 1999), key elemets of which will be presented in the following. (Drucker, 1999) uses the terminology "training pattern". It it assumed this is the same as the feautres in the training set. It also uses the termionology "regression machine". It is assumed that this is equivalent to what we have previously defined as a set of trees.

We begin with a training set with N_1 features. Each feature is weighted by a weight $w_i, \quad i = 1, 2, \cdots, N_1$. Then we run the algorithm:

1) Let the probability of a training sample be in the training set be $P_i = \frac{w_i}{\sum_i w_i}$. We use this probability to draw N_1 samples, with replacement, from the training set.
2) Contruct a regression machine, and train it: $h_t : x \rightarrow y$
3) Make a prediction on the training data $y_i^{(p)}(\mathbf{x}_i)$
4) Calculate the loss. We have a few alternatived to choose from:
Let $D = sup|y_i^{(p)}(\mathbf{x}_i) - y_i|$, then we have

Linear:

$$L_i = \frac{|y_i^{(p)}(\mathbf{x}_i) - y_i|}{D}$$

Suare law:

$$L_i = \frac{|y_i^{(p)}(\mathbf{x}_i) - y_i|^2}{D^2}$$

Exponential:

$$L_i = 1 - exp(\frac{-|y_i^{(p)}(\mathbf{x}_i) - y_i|}{D})$$

5) Find the average loss $\bar{L} = \sum_{i=1}^{N_1} L_i p_i$
6) Define a confidence parameter $\beta = \frac{\bar{L}}{1-\bar{L}}$. A lower β indicates a higher confidence in a prediction

7) Update the weights with $w_i \rightarrow w_i \beta^{1-L_i}$

8) Find the weighted median:

$$h_f = inf\{y \in Y : \sum_{t:h_t \leq y} log(1/\beta_t) \geq \frac{1}{2} \sum_t log(1/\beta_t)\}$$

Repeat 1-8 until the average loss $\bar{L} < 0.5$

So with the hypothesis h_f from step 8 we can make a prediction.

2.7 Error, bias, and variance

Finally we need to discuss error, bias, variance, and more. We will start with cost functions.

The perhaps simplest cost function we can come up with is the mean squared error (mse)

$$mse = C(\beta) = \frac{1}{n} \sum_{i=1}^{n} (y - X\beta)^2$$

The use of this cost function is of great benefit, when we're after mathematical simplicity. Just look at linear regression. The whole method hinges upon the solution to minimizing the mse. From the mse we can even derive the bias variance trade-off (Hagen, 2020b):

$$\mathbb{E} = [(y - \tilde{y})^2] = \frac{1}{n}\sum_i (f_i - \mathbb{E}[\tilde{y}])^2 + \sum_i (\tilde{y_i} - \mathbb{E}[\tilde{y}])^2 + \sigma^2 \tag{20}$$

In (15) we have assumed the target $y = f + \epsilon$, where f is some function and ϵ is some normally distributed noise σ^2 in our data. $\tilde{y}$ is our model.

What (15) tells us is that the expected value of the suared difference between the targets and the model is a combination of the variance (first term in (15)), the bias (second term) and σ^2.

This leads us to the bias variance trade-off (BVT). BVT tells us that we can dencrease the variance, by increasing the bias. Or more generally: as one increases the complexity of model the variance drops, but this will increase the bias. This is pretty much the definition of a trade-off, and plagues many aspects of machine learning.

This trade-off is one of the reasons boosting was "invented". Boosting is able to both lower variance and bias.

Method

Preliminary remark 1: For all algorithms, with the exeption of the neural network, we use ScikitLearn.

We have a dataset made up of 9 features: Overall rank, Country or region, Score, GDP per capita, Social support, Healthy life expectancy, Freedom to make life choices, Generosity, Perceptions of corruption. This data is gathered from 156 different countries (or regions). We can immediatly dispense with Overall Rank, and Country or Region, as we are not much intrested in these, but the (happiness) score

Our target data is the Score, so we separate this data from the set, so we can see if we can find a model that, based upon the other features, can predict the happiness score. What we'll do next is a principal component analysis to see if there are any features not correlating much with the targets at all. If this is the case, we can remove this feature from the dataset without much detriment to the error of the predictions we will make.

Preliminary remark 2: For some of the more computatially heavy algorithms we will collect the execution time from first training on a "full" dataset, and the a "reduced" dataset. That is, in the case of PCA showing us that one or more principal component can be removed from the "full" dataset, we remove it/ them, leaving us with the "reduced" dataset. Aside from that, we will for most algortihms train on both the reduced and full dataset as to compare the error.

The modelling will first be done with Linear Regression and cross-validation. We use with and without regularization (L1 and L2). With L1 regression we initially wanted to do two different approaches: ScikitLearn, and code I've written, including mini-batch gradient descent for approximating the minimum of the cost function. This got reduced to only ScikitLearn, due to time constraints, as well as linear regression not being the main focus of this project. ScikitLearn uses coordinate descent, a method we'll not further investigate. We assess the mean squared error of the model we've made, and see if the result is satifactory.

Then we use the feed forward neural network to see if we get an impoved model. We will test the different activation functions (9), (10), and (11), and gather the execution time of the code. We compare these results (MSE) to the linear regression results.

We go on to trees. First we'll grow a single decision tree, and see how the depth of the tree affects the predictions.

After this we'll expand into random forrests, bagging and boosting. We will consider what tree depth, and loss functions has to say for our predictions. We will also compare the bias variance trade-off in regards to bagging and the two different boosting methods we've outlined.

Results

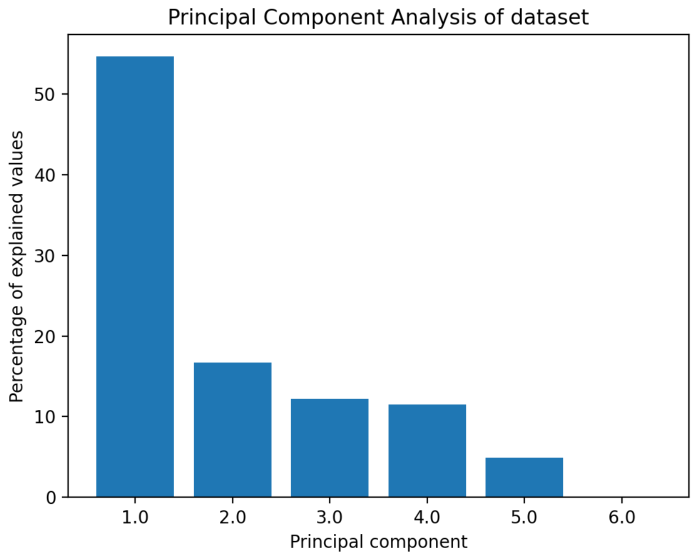


Figure 1: Percentage variance contribution of each principal component

Where $1, 2, \cdots, 6$ represents principal components $PC1, PC2, \cdots, PC6$, and their representations are presented in table 1

PC1	PC2	PC3	PC4	PC5	PC6
GDP per capita	Social support	Healthy life expectancy	Freedom to make life choices	Generosity	Perceptions of corruption

Table 1: List of principal components

We se that $PC6$ has almost no contribution to the variance at all, and that GDP per capita is responsible for over 50% of it. Hence we plot the happiness score as a function of GDP per capita, as shown in figure 2

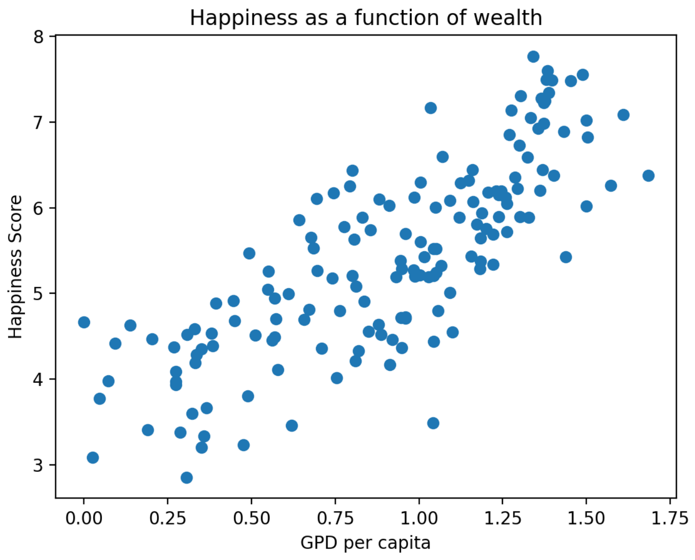


Figure 2: Happiness score as a function of GDP per capita

We do the first regression analysis without regularization, both with and without PC6. The results are presented in table 2:

Train MSE OLS full dataset	Test MSE OLS full dataset	Training MSE OLS reduced dataset	Test MSE OLS reduced dataset
0.40	0.43	0.43	0.34

Table 2: Train and test MSE with and without PC6

Reducing the dimensionality does not drastically impact the error in the prediction and test, as is evident from table 2.

The results from the Ridge regression with cross validation is presented in table 3.

Train MSE Ridge full dataset	Test MSE Ridge full dataset	Training MSE Ridge reduced dataset	Test MSE Ridge reduced dataset
0.43	0.38	0.39	0.48

Table 3: test and training results from Ridge regression using cross validation.

To obtain the data in table 3 we have used ScikitLearn's RidgeCV method with $alphas = logspace[-5, 1, 100]$. That is: $\alpha = 10^{-5}, \cdots, 10^{1}$

We used ScikitLearn's LassoCV with $eps = logspace(-7, -1, 40)$, and $n\_alphas = 500$. We have let ScikitLearn choose the different alpha-values, and just asserted the number of them. We've plotted the MSE as a function of eps, as presented in figures 3 and 4.

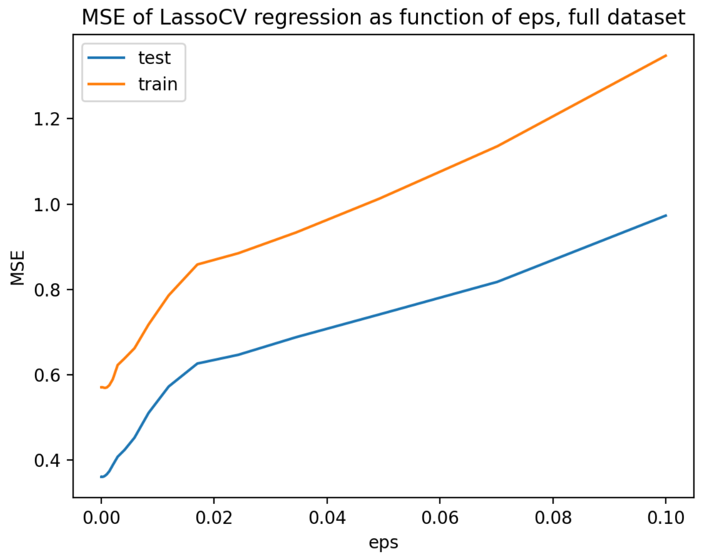


Figure 3: MSE in Lasso regression with cross validation as a function of eps, on the full dataset

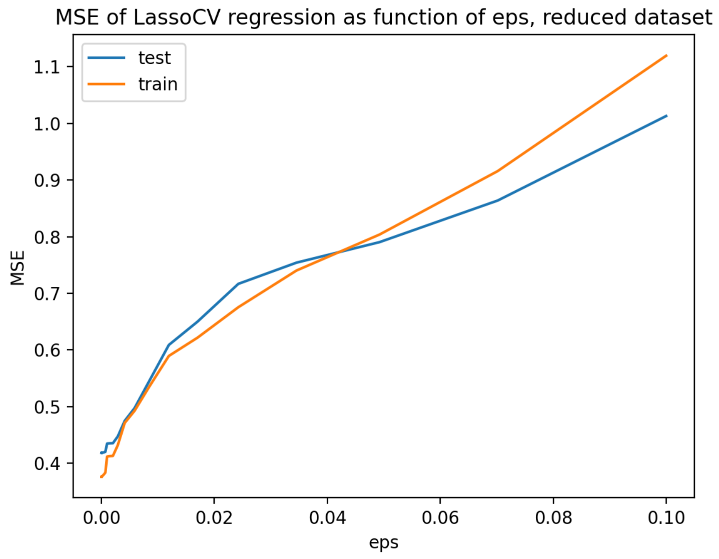


Figure 4: MSE in Lasso regression with cross validation as a function of eps, on the reduced dataset

We run the network three times on the full dataset (dataset including PC6), each time with a different activation function. We repeat this with the reduced dataset. We also check the execution time for each run, because neural networks (at least the one I've written) are somewhat slower than other mothods. The results are presented in table 3.

Activation function	Dataset	MSE train	MSE test	Execution time [s]

Activation function	Dataset	MSE train	MSE test	Execution time [s]
Sigmoid	Full	1.39	1.07	4.87
RELU	Full	2.11	2.01	10.70
Leaky RELU	Full	2.26	1.96	17.15
Sigmoid	Reduced	1.26	1.65	4.64
RELU	Reduced	1.77	2.08	7.50
Leaky RELU	Reduced	2.16	1.69	5.96

Table 4: training MSE, test MSE, and execution time for all three activation functions, using both the full and reduced dataset.

In the case of the full dataset, the neural network outperforms the regression methods when using the Sigmoid activation function. The RELU and LeakyRELU seems to perform somewhat worse, though they are slightly more time efficient.

We ran the neural network for 400 iterations, with a learning rate of 10^{-5} and a regularization of 1. We also divided the training set up into 50 mini-batches. The network used one hidden layer, the same size as the input layer: 6 nodes in the case of the full dataset, and 5 nodes in the case of the reduced dataset. These values and parameters were obtained somewhat ad-hoc, and based on experience. We ran the neural network for a variety different values for these parameters, and the values presented seemed to be roughly the best. Future work could be to device an optimization scheme to find the best parameters, but serious efforts must be put in to optimize the execution time of the network before such work can begin.

For the regression tree we usen ScikitLearn's method. We ran it for depths ranging between 1 and 14. The number 14 is a result of us allowing a minimum leaf size of 1. This resulting in that the tree can't get any deeper, unless we want leaf sizes of zero, which would make no sense.

We plotted the MSE against the depth, as presented in figures 5 and 6.

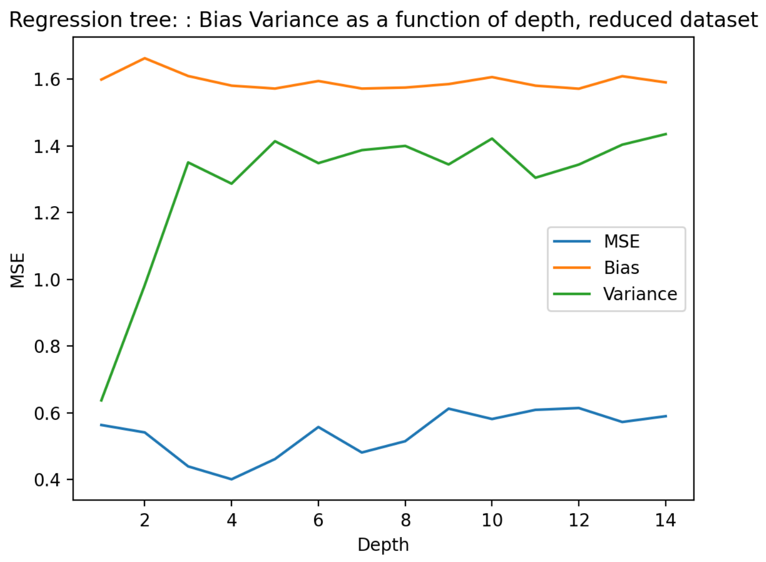


Figure 5: Regression tree MSE as a function of depth, full dataset

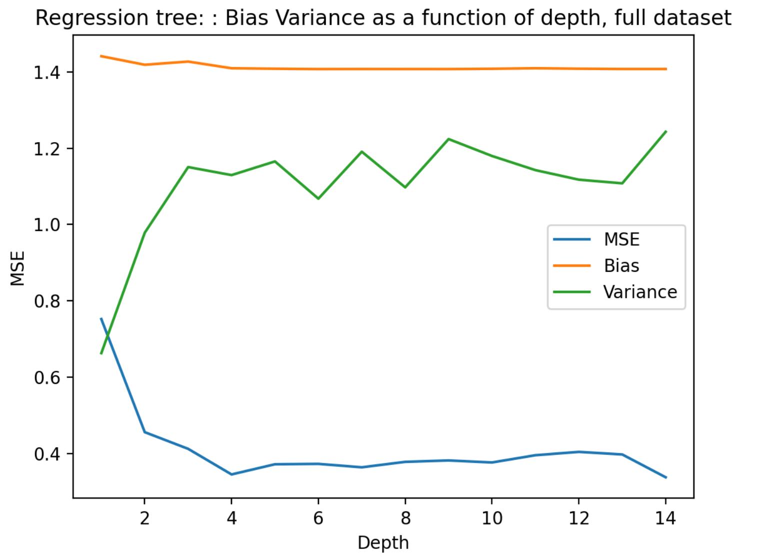


Figure 6: Regression tree MSE as a function of depth, reduced dataset

We also registered the execution time, as presented in table 5

Method	Dataset	Execution time [s]
Regression Tree	Full	0.011
Regression Tree	Reduced	0.013

Table 5: Execution time of regression tree for full, and reduced dataset

We can see from both figure 4 and 5 that the error is the lowest when the depth is one. This might be an indication of our data being somewhat tightly clustered in one region, and the initial partition of the feature space contains most of the values. Overfitting seems to occur immediately.

We can see a somewhat similar result in figure 7 and 8, where we plotted the MSE of a random forest against the depth of the trees, though the error is altogether considerably reduced.

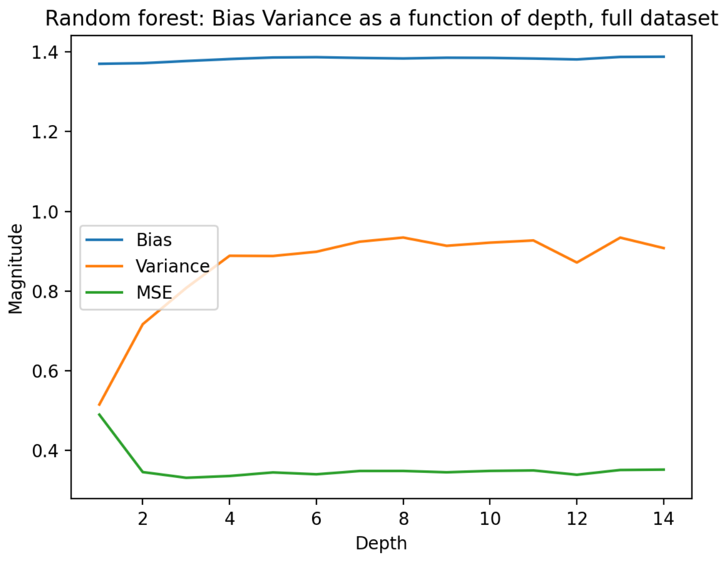


Figure 7: Random forest MSE as function of depth of trees, full dataset

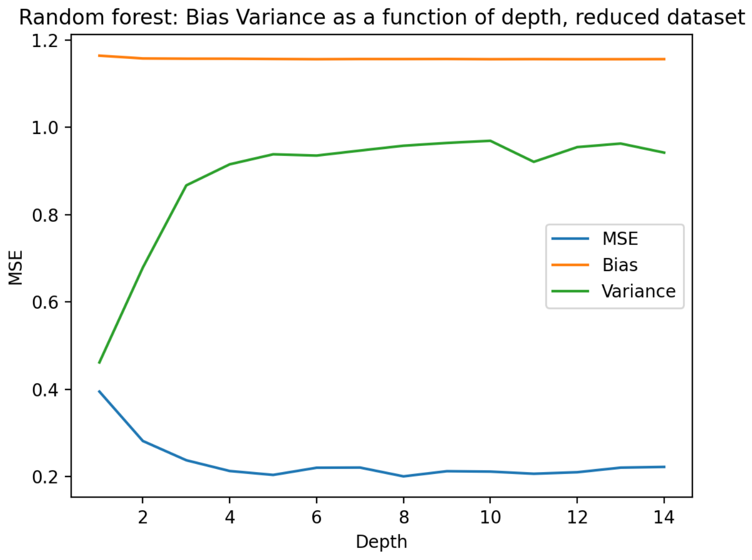


Figure 8: Random forest MSE as function of depth of trees, full dataset

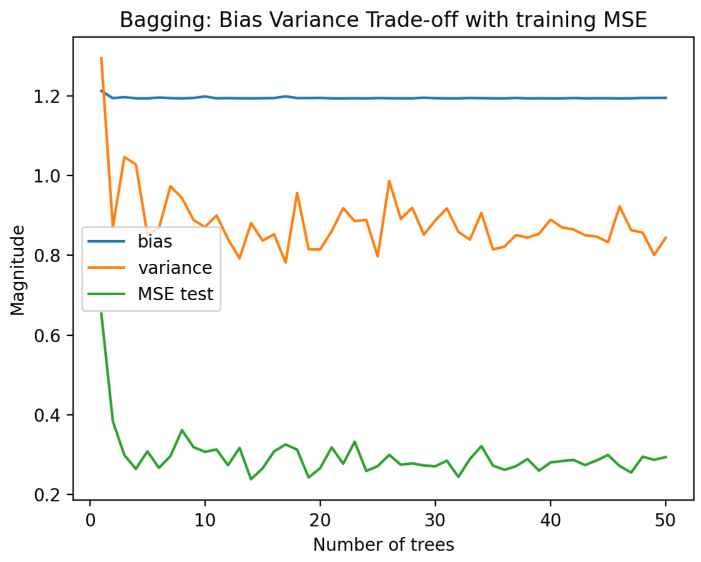


Figure 9: Bias and variance of bagging

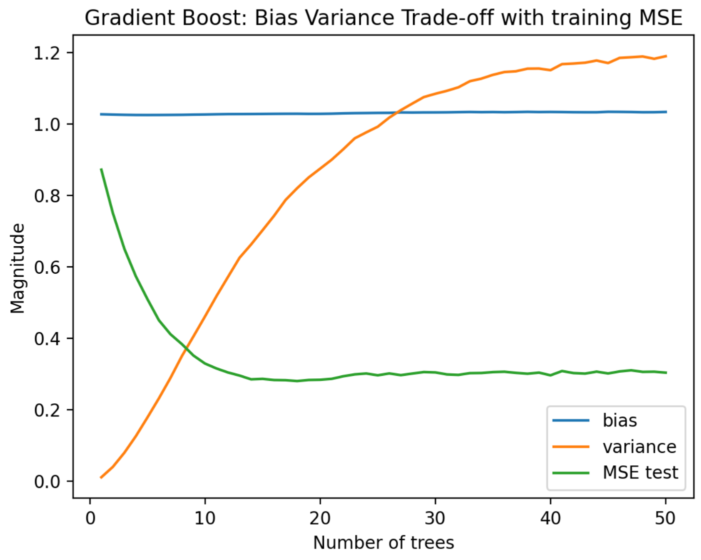


Figure 10: Bias and variance of gradient boosting

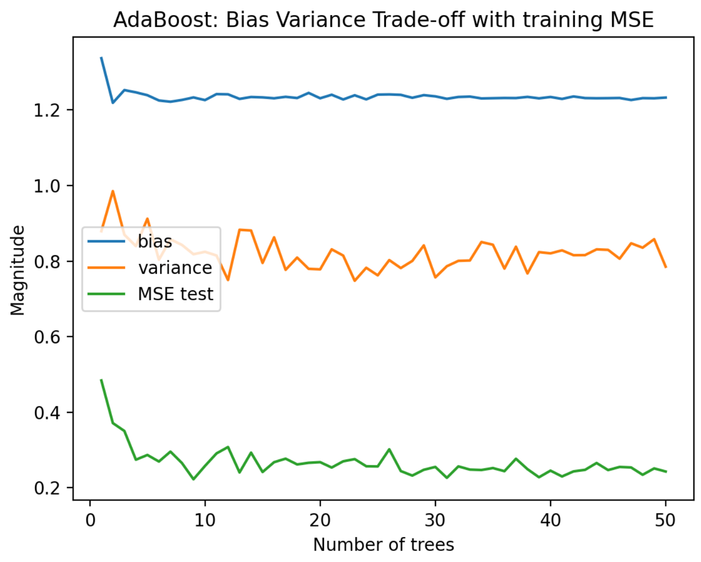


Figure 11: Bias and variance of AdaBoost

In figures 9 to 11 we have used trees of depth 3, and a learning rate of 0.1. These are among quite a few parameters we could tune.

Discussion

First thing that bear mentioning is that, with the exception of the neural network, there were no instances of feature scaling. In some cases scaling lead to MSE of around 30. Every method was tested with and without scaling, and allthough no data of that is presented (I can only apologize. Try the code yourself.), every algorithm except the neural network performed best with un-scaled data.

The first rather interesting observations we can make is that Perceptions of corruption (PC6) has very low correlation with the happiness score. We were able to create models both with and without this feature, both cases yielded reasonably accurate models.

From figure 2 we observe a clear linear relationship between GPD per capita and happiness score.

In the OLS and ridge regression case, the errors seemed rather indifferent to PC6 as obsereved from tables 2 and 3. We could perhaps in the future try to fit models of increasing complexity on this data, and do a bias and variance analysis. By Increasing complexity it is meant that we first do an analysis on PC1, then on PC1 and PC2, etc, and compare the models against their complexities.

A problem I (to my shame) have failed to address thus far is the correlation between some of the features in our dataset. Features such as Social Support, Life Expectancy, corrolates highly with GDP per capita. This might be problematic, and should be taken into account when modelling the data. Unfortunately,this was not done. So bear that in mind as I will proceed to further discuss our results, never mentioning these correlating features again. Except perhaps once or twise.

Lasso regression is the first method that shows an obvious "preferance" for which dataset to use.

The full dataset categorically provides lover error than the reduced one, yet we fail to reproduce the accuracy, which is not really surprising.
OLS is based upon finding the inverse of $X^T X$, and provided this inverse exists, OLS is the best method for linear regression. As it happens, in our case $X^T X$ is non-singular. This means that we don't need to find the pseudo-inverse, or use regularization parameters. The best solution for linear regression is already available to us.

By far the worst performing model was the neural network. I suspect this might be due to a couple of reasons.
1) We did not spent sufficient time tuning the hyperparameters. We did it quick and easy ("by hand") and got not too bas results, allthough comparatively bad.
2) This was the only algorithm written by the author of this "paper". Every other method used has been from Scikit-learn, and it's not too far fatched to believe they've spent significantly more time optimizing their algorithms. They're also a team of professionals. I'm neither a team nor a professional.

As mentioned in the beginning of this section, the neural network, for some reason, benefited from feature scaling. Not only did this scaling improve the error, by almost a factor of 2, but it reduced the execution time by around the same factor in some instances. Why this might be is a complete mystery and warrants further research.

The perhaps most surprising result was the lack of overfitting in the regression tree. In both figures 5 and 6 we observe quite stable prediction error, even when we grow the tree to maximum possible depth. This is rather unexpected behaviour, and could be a result of the correlating features I said I wouldn't mention again. As for the full dataset compared to the reduced one, we observe from figures 5 and 6 the the tree performs better with the full dataset. This might stem from the fact that PC6 contributes to some stochasticity to the partitions in a way that makes overfitting harder, but this is purely conjecture.

It's also apparent from the tree-models that we have a certain amount of bias in our data.

As for performance, the regression trees were on par with OLS. The regression tree trained on the full dataset even did a little better than the regression methods, implying that our data is not perfectly linear. Which is only what one would expect, really.

The random forests were able to reduce the bias value by 0.2, something we would expect. This is due to the randomly selected subset of features, which helps to eliminate a certain amount of the bias in the data. We were also able to drive the error down quite a bit compared to the regression trees. It was almost halved. This is further evidence to the idea that an ensemble of weak learners are better than one strong learner.

As one would perhaps excpect, bagging did not further decrease the bias. This is because no randomness is introduced in selecting the features of a tree, resulting in bias being "carried over".

In fact: when speaking of bias, easly the best performing model was the gradient boosted one. It manages to reduce it from 1.4 in the regression tree, all the way down to around 1. The sacrifice for this great reduction in bias is obviously the variance. The variance skyrockets as from the moment we introduce a hint of complexity. Also the MSE converges rather slowly.

With these considerations in mind, AdaBoost seem to be a good compromise between low MSE, acceptable bias, and stable variance. This is of course a an opinion unique to this particular dataset. There might be cases where high bias is acceptable, but the variance must be low, or vice versa. In our case, we're quite content with our results as they stand.

Conclution

The results presented herein demonstrates clearly the capability of ensamble methods: sets of weak learners. In the context we've used them, they far outperform strong learners, both in execution time and predictive power. And also: Apparently money can, in fact, buy happiness.

Refrences

[1] United Nation (2019) https://www.kaggle.com/unsdsn/world-happiness (https://www.kaggle.com/unsdsn/world-happiness)
[2a] Hagen, M. (2020, November, 13) Project 2. https://github.com/magnuhag/Project2_FYS-STK3155 (https://github.com/magnuhag/Project2_FYS-STK3155)
[2b] Hagen, M. (2020, October, 23) Project 1. https://github.com/magnuhag/Prosjekt1_fys-stk-3155 (https://github.com/magnuhag/Prosjekt1_fys-stk-3155)
[3] Wikipedia (2020, December, 9) https://en.wikipedia.org/wiki/Principal_component_analysis (https://en.wikipedia.org/wiki/Principal_component_analysis)
[4] Hjorth-Jensen, M. (2020, October, 26) https://compphysics.github.io/MachineLearning/doc/pub/week43/html/week43.html (https://compphysics.github.io/MachineLearning/doc/pub/week43/html/week43.html)
[5] Wikipedia (2020, December, 8) https://en.wikipedia.org/wiki/Moore–Penrose_inverse (https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse)
[6] Hastie, T., Tibshirani, R., Friedman, J. (2009) The Elements of Statistical Learning
[7] Drucker, H. (1999, August) https://www.researchgate.net/profile/Harris_Drucker/publication/2424244_Improving_Regressors_Us
Regressors-Using-Boosting-Techniques.pdf
(https://www.researchgate.net/profile/Harris_Drucker/publication/2424244_Improving_Regressors_Us
Regressors-Using-Boosting-Techniques.pdf)