MATH533 A3 - Random Forests, Variable Importance, and Mean Decrease in Accuracy

Gulliver Häger, Evelyn Hubbard, Michael Montemurri

November 2024

Contents

1 Introduction to the Random Forest

We consider a setting where we have random variables $(\boldsymbol{x}, y) \in \mathcal{X} \times \mathbb{R}$ and we wish to predict the value of $y \in \mathbb{R}$, given the random vector $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^p$. We make the additional assumption that $\mathbb{E}\left[y^2\right] < \infty$. An elementary result of regression analysis is that the regression function $m : \mathbb{R}^p \to \mathbb{R}$ defined as $m(\boldsymbol{x}) = \mathbb{E}\left[y \mid \boldsymbol{x}\right]$ is the optimal such function in the sense that it minimizes the $\mathcal{L}_2$-distance between the predicted value $m(\boldsymbol{x})$ and the true value of y.

If the joint probability distribution of $\boldsymbol{x}$ and y is not known, as is the case in many practical situations, it is not possible to obtain the true regression function. Instead, we must define an estimate of the regression function based on a given sample data of $n \in \mathbb{N}$ independent and identically distributed (IID) random variables, denoted $\mathcal{D}_n := \{(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n)\}$, sampled from the same distribution as $(\boldsymbol{x}, y)$. For a sample of size n, m_n is the estimate of m from $\mathcal{D}_n$. Ideally, we would like to construct a sequence of estimators $\{m_n\}_{n\in\mathbb{N}}$ which is *mean squared error consistent*, meaning that for any error tolerance in $\mathcal{L}_2$-distance, we can always find a large enough $n \in \mathbb{N}$, for which m_n is within that tolerance to m. Equivalently, we can state this as:

$$\lim_{n\to\infty} \mathbb{E}\left[(m_n(\boldsymbol{x}) - m(\boldsymbol{x}))^2\right] = 0 \tag{1}$$

At this point, there are several ways to construct m_n that estimates the regression function. *Linear regression* assumes a parametric model where $\boldsymbol{\beta}$ is the unknown parameter and assumes that $m_n(\boldsymbol{x})$ is linear in $\boldsymbol{\beta}$.

Alternatively, we may proceed in a non-parametric approach such as the *Random Forest* (RF) algorithm. RFs may refer to a variety of ensemble methods for combining the results of many random decision trees to make a prediction. The discussion in this paper, motivated by Biau and Scornet's RF survey paper [1], will use RF to refer to Breiman's algorithm [2].

1.1 Regression Trees

A regression tree is an estimation of the regression function based on some sampled data $\mathcal{D}_n$, and thus a function from $\mathbb{R}^p$ to $\mathbb{R}$. Let $m_n(\boldsymbol{x}; \mathcal{D}_n)$ be the prediction of the regression tree for $\boldsymbol{x} \in \mathcal{X}$. Each node in a regression tree corresponds to a a hyper-rectangular cell in $\mathcal{X} \subseteq \mathbb{R}^p$ and together all cells corresponding to the leaves in the regression tree form a partition of $\mathcal{X}$. Any sampled point $\boldsymbol{x} \in \mathcal{X}$ therefore falls into a cell corresponding to a leaf in the tree, and the predicted value of the regression tree for $\boldsymbol{x}$ is the average value of all data points from $\mathcal{D}_n$ which fall into the same cell as $\boldsymbol{x}$. The root node of the regression tree corresponds to the sample space $\mathcal{X}$ itself.

To construct a regression tree, we recursively split the region of each node until every node contains less than $s \in [n]$ data points from $\mathcal{D}_n$, where s is a parameter chosen before the construction of the tree. At each node, the cell is split along a hyperplane defined by one so called splitting covariate being constant. The choices of the splitting covariate and value at which we split are made to maximize the *CART-split criterion*. The two produced regions are then assigned to two new nodes in the tree which are the children of the original parent node.

The CART-split criterion intuitively aims to split a cell A containing some data points in such a way that the data in each of the resulting two cells from the split are as similar as possible in value in the variable of interest. To define the splitting criterion, first let $\mathcal{Z}_j$ be all midpoints in the feature j for all points of $\mathcal{D}_n$ which are in the cell. For a cell A, we define a *cut* as a pair

$(j, z) \in [p] \times \mathcal{Z}_j$ which splits A with respect to the co-variate $j \in [p]$ at $z \in \mathcal{Z}_j$ into the two new cells $A_L := \{\boldsymbol{x} \in A : x^{(j)} < z\}$ and $A_R := \{\boldsymbol{x} \in A : x^{(j)} \geq z\}$. We define C_A as the set of all possible cuts, (j, z), of cell A, and $N_n(A)$ the number of data points of $\mathcal{D}_n$ falling in A.

For any given cell A, the empirical variance of all $\{y_i\}_{i \in [n]}$ which have their corresponding co-variates are in A is:

$$\frac{1}{N_n(A)} \sum_{\substack{i \in [n] \\ \boldsymbol{x}_i \in A}} (y_i - \bar{y}_A)^2 \tag{2}$$

For any cut $(j, z) \in \mathcal{C}_A$, let $\bar{y}_{A_L}$ and $\bar{y}_{A_R}$ be the sample average of all points in A_L and A_R, respectively. We can now look at how similar the values in each of the new cells are to each other by looking at the modified sample variance defined as:

$$\frac{1}{N_n(A)} \sum_{\substack{i \in [n] \\ \boldsymbol{x}_i \in A}} (y_i - \bar{y}_{A_L} \mathbb{1}_{\{\boldsymbol{x}_i \in A_L\}} - \bar{y}_{A_R} \mathbb{1}_{\{\boldsymbol{x}_i \in A_R\}})^2 \tag{3}$$

If we think of the cut (j, z) as an attempt to partition A to group all points in A in terms of their similarities in their values of the variable of interest, then the difference in these two conditional variances acts as a measure of the effectiveness of the cut. This is, in fact, the CART-split criterion, which is defined as:

$$L_{\text{reg},n}(j, z) := \frac{1}{N_n(A)} \sum_{\substack{i \in [n] \\ \boldsymbol{x}_i \in A}} \left[(y_i - \bar{y}_A)^2 - (y_i - \bar{y}_{A_L} \mathbb{1}_{\{\boldsymbol{x}_i \in A_L\}} - \bar{y}_{A_R} \mathbb{1}_{\{\boldsymbol{x}_i \in A_R\}})^2 \right] \tag{4}$$

Since our goal is to produce a cut in which the two resulting cells contain points which are closer to each other on average, we want to maximize $L_{\text{reg},n}$. For a given cell A, we let (j^*, z^*) be an optimal cut such that $L_{\text{reg},n}$ is maximized. Notice that such a cut is not necessarily unique, and we have to employ some form of tie-breaking procedure to select the cut which will be used to construct the tree. In notation, we have that for a given cell A:

$$(j^*, z^*) \in \underset{(j,z) \in \mathcal{C}_A}{\arg\max} \, L_{\text{reg},n}(j, z) \tag{5}$$

If we let $A_n(\boldsymbol{x}; \mathcal{D}_n)$ be the cell containing $\boldsymbol{x}$, we can rewrite the predicted value of the regression tree at $\boldsymbol{x} \in \mathcal{X}$ as:

$$m_n(\boldsymbol{x}; \mathcal{D}_n) = \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_n} \frac{\mathbb{1}_{\{\boldsymbol{x}_i \in A_n(\boldsymbol{x}; \mathcal{D}_n)\}} y_i}{N_n(A_n(\boldsymbol{x}; \mathcal{D}_n))} \tag{6}$$

Note that for the construction of a regression tree, we take $\mathcal{D}_n$ as input along with the parameter $s \in [n]$.

1.2 Constructing a Random Forest

At a high level, the RF algorithm consists of aggregating the results of $M \in \mathbb{N}$ random regression trees. A random regression tree is constructed similarly to a regression tree, with two differences. The first difference is that the input data for the random regression tree, call it $\mathcal{D}_n^*$, is selected as a bootstrap sample from $\mathcal{D}_n$ of size a_n where $a_n \in [n]$ is a chosen parameter. We can let Θ be some random variable which is used to select this bootstrap sample. The second difference is that for every

cut made, we only consider a uniformly random subset $\mathcal{M}_{\text{try}} \subseteq [p]$ of size m_{try} of all co-variates, where $m_{\text{try}} \in [p]$ is also a chosen parameter. The chosen cut (j^*, z^*) at any node with corresponding cell A will then by calculated as:

$$(j^*, z^*) \in \underset{\substack{(j,z) \in \mathcal{C}_A \\ j \in \mathcal{M}_{\text{try}}}}{\arg\max} L_{\text{reg},n}(j, z) \tag{7}$$

We modify our notation from the previous section to account for this randomness, by letting $\mathcal{D}_n^*(\Theta)$ be the bootstrap sample used to construct the tree and $A_n(\boldsymbol{x}; \Theta, \mathcal{D}_n)$ the cell which contains $\boldsymbol{x} \in \mathcal{X}$. We then get that the predicted value for $\boldsymbol{x}$ for the random regression tree is:

$$m_n(\boldsymbol{x}; \Theta, \mathcal{D}_n) = \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}_n^*(\Theta)} \frac{\mathbb{1}_{\{\boldsymbol{x}_i \in A_n(\boldsymbol{x}; \Theta, \mathcal{D}_n)\}} y_i}{N_n(A_n(\boldsymbol{x}; \Theta, \mathcal{D}_n))} \tag{8}$$

To create the RF, we then generate M random regression trees, where for any $j \in [M]$ we can let $m_n(\boldsymbol{x}; \Theta_j, \mathcal{D}_n)$ be the prediction of the jth random regression tree at $\boldsymbol{x}$. Here $\{\Theta_j\}_{j \in [M]}$ are IID random variables used to select the bootstrap sample each tree is constructed using.

The RF algorithm combines the prediction of each tree by averaging their predicted values. The estimated value for $\boldsymbol{x}$ of the RF is:

$$m_{M,n}(\boldsymbol{x}; \Theta_1, \ldots, \Theta_M, \mathcal{D}_n) := \frac{1}{M} \sum_{j \in [M]} m_n(\boldsymbol{x}; \Theta_j, \mathcal{D}_n) \tag{9}$$

Since each Θ_j is IID we have that $\{m_n(\boldsymbol{x}; \Theta_j, \mathcal{D}_n)\}_{j \in \mathbb{N}}$ is a collection of IID random variables and only dependent on Θ_j, conditional on knowing $\mathcal{D}_n$. With the additional assumption that these random variables have a finite expectation, we get by Kolmogorov's strong law of large numbers that:

$$\lim_{M \to \infty} m_{M,n}(\boldsymbol{x}; \Theta_1, \ldots, \Theta_M, \mathcal{D}_n) \xrightarrow{\text{a.s.}} \mathbb{E}\left[m_n(\boldsymbol{x}; \Theta, \mathcal{D}_n)\right] \tag{10}$$

We can thus let

$$m_{\infty,n}(\boldsymbol{x}; \mathcal{D}_n) := \mathbb{E}\left[m_n(\boldsymbol{x}; \mathcal{D}_n)\right] \tag{11}$$

be our idealized infinite forest estimator. Given enough computational power, we can always get within an arbitrary accuracy of $m_{\infty,n}$. Thus, for the RF model we can consider this infinite forest estimator in our theoretical discussion.

Notice that the expectation above is taken with respect to the random variable Θ, conditional on $\mathcal{D}_n$. The expectation is conditional on $\mathcal{D}_n$ since when we are constructing our RF, we are given the n data points $\mathcal{D}_n$ already, but will need to realize the random variables $\{\Theta_j\}_{j \in [M]}$.

We emphasize that the construction of a random regression tree, and thus also a RF, has the following parameters. For a discussion of the tuning of these parameters in practice, see Biau's survey paper [1].

- $a_n \in [n]$: the size of the number of samples from $\mathcal{D}_n$ used in the construction of each tree
- $m_{\text{try}} \in [p]$: the size of the random subset of the features considered for splitting at each node
- $s \in [a_n]$: the number of points at which we stop splitting a cell if it contains less than s points

1.3 Breiman's Algorithm

In the previous two sections, we described the procedure of how to create a random tree based on the CART-split criterion. Recall that the RF algorithm estimate is the average value of the prediction of $M \in \mathbb{N}$ such random regression trees. In this section, we give the pseudo-code for the complete algorithm on how to produce the estimator $m_{M,n}(\boldsymbol{x}; \Theta_1, \ldots, \Theta_M, \mathcal{D}_n)$. This pseudo-code is close to that of Biau[1].

In Breiman's original paper [2], suggests fully growing the trees without any pruning, i.e. setting $s = 1$.

Algorithm 1: Breiman's Random Forest Algorithm

Input: Number of trees $M \in \mathbb{N}$. Training data $\mathcal{D}_n$. Tree creation parameters $a_n \in [n]$, $m_{\text{try}} \in [p]$, $s \in [a_n]$

Output: The Random Forest Estimator $m_{M,n}(\boldsymbol{x}; \Theta_1, \ldots, \Theta_M, \mathcal{D}_n)$

```
for j ∈ [M] do
    Select a_n points uniformly at random, with replacement, from D_n. Call these points
     D*_n(Θ_j). Only these points are used in the following steps.
    Let cellsToSplit = new queue(X)
    Let finalCellList = []
    while cellsToSplit ≠ [] do
        Let A = cellsToSplit.peek()
        if A contains less than s points (or if all points in A are equal) then
            cellsToSplit.pop()
            finalCellList.append(A)
        else
            Select a subset M_try ⊆ [p] uniformly at random with cardinality m_try
            Split A into A_L and A_R by maximizing the CART-criterion
            cellsToSplit.append(A_L, A_R)
    For a given x ∈ X, let m_n(x; Θ_j, D_n) be the average of all y_i for all points which fall
     into the same cell as x in the partition defined by the elements of finalCellList
For a given x ∈ X, return the average of m_n(x; Θ_j, D_n) over all values j ∈ [M]
```

2 Variable Importance: MDA Metric

The RF algorithm can be used to quantify the significance of features. The Mean Decrease Accuracy (MDA) metric, as proposed in Breiman's 2001 paper [2] assesses a feature's impact on the prediction accuracy of the RF. The MDA of a feature represents the difference in the RF prediction error on out-of-bag (OOB) samples and the re-evaluated prediction error when the values of that feature for the same OOB samples are randomly permuted. In this context, an OOB sample refers to one not included in the bootstrapped dataset, $\mathcal{D}_n^*$. Conceptually, if a variable is not important, permuting its values should not impact the RF prediction accuracy and the MDA should be small, relative to other features [1].

We will consider the following additive regression model $y = m(\boldsymbol{x}) + \epsilon$, where $\boldsymbol{x} = (x^{(1)}, ..., x^{(p)})$ is a vector of random variables (representing the features) and $m(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}]$. Because the model is

additive, the regression function can be expressed as:

$$y = \sum_{j=1}^{p} m_j(x^{(j)}) + \epsilon$$

where $m(\boldsymbol{x}) = [m_1(\boldsymbol{x}), ..., m_p(\boldsymbol{x})]$

2.1 Theoretical MDA

Using the model explained above, the true MDA of a feature $x^{(j)}$ is

$$\text{MDA}(x^{(j)}) = \mathbb{E}\left[(y - m(\boldsymbol{x}^{j'})^2\right] - \mathbb{E}\left[(y - m(\boldsymbol{x}))^2\right]$$

where $\boldsymbol{x}^{j'} = (x^{(1)}, ..., x^{(j')}, ..., x^{(p)})$ is a version of $\boldsymbol{x}$ with $x^{(j')}$ an independent replicate of $x^{(j)}$. For a sample D_n (with size n), a replicate is produced by permuting the n values of $x^{(j)}$ (one from each sample).
Following results in [3], we show that if we make the following assumptions about the model, the theoretical MDA can be found using only correlation information from $(\boldsymbol{x}, y)$.

Assumption 2.1

1. *$\boldsymbol{x}$ and ϵ are centered Normal variables.*
2. *The covariance matrix $C = \left[Cov(x^{(j)}, x^{(k)})\right]_{1 \leq j,k \leq p}$ has the following form:*

$$C = (1-c)I_p + c(1, ..., 1)(1, ..., 1)^\top = \begin{bmatrix} 1 & c & \dots & c \\ c & 1 & & c \\ \vdots & & \ddots & \\ c & \dots & c & 1 \end{bmatrix}$$

   where $c \in (0, 1)$ and I_p is the identity matrix.

3. *$Cov(x^{(j)}, y) = \tau_0$ for all $j \in \{1, ..., p\}$.*

An equivalent way to note these assumptions is:

$$(\boldsymbol{x}, y) \sim \mathcal{N}\left(0, \begin{bmatrix} C & \tau \\ \tau^\top & \sigma^2 \end{bmatrix}\right) \tag{1}$$

where $\tau = (\tau_0, ..., \tau_0)^\top \in \mathbb{R}^p$ and $\sigma^2 \in \mathbb{R}$.

Theorem 2.1 *[3, Proposition 3]*
If assumption 2.1 are satisfied, the theoretical MDA of feature $x^{(j)}$ is

$$MDA(x^{(j)}) = 2\left(\frac{\tau_0}{1 - c + pc}\right)^2 \tag{2}$$

Proof.
We start from the definition of MDA.

$$\begin{aligned}
\text{MDA}(x^{(j)}) &= \mathbb{E}\left[(y - m(\boldsymbol{x}^{j'})^2\right] - \mathbb{E}\left[(Y - m(\boldsymbol{x})^2\right] \\
&= \mathbb{E}\left[(y - m(\boldsymbol{x}) + m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'})^2\right] - \mathbb{E}\left[(y - m(\boldsymbol{x}))^2\right] \\
&= \mathbb{E}\left[(y - m(\boldsymbol{x}))^2\right] + 2\mathbb{E}\left[(y - m(\boldsymbol{x}))(m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'}))\right] + \mathbb{E}\left[(m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'})^2\right] - \mathbb{E}\left[(y - m(\boldsymbol{x}))^2\right] \\
&= \mathbb{E}\left[\epsilon m(\boldsymbol{x})\right] - \mathbb{E}\left[\epsilon m(\boldsymbol{x}^{j'})\right] + \mathbb{E}\left[(m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'}))^2\right] \\
&= \mathbb{E}\left[m(\boldsymbol{x})\mathbb{E}\left[\epsilon|\boldsymbol{x}\right]\right] - \mathbb{E}\left[\epsilon\right]\mathbb{E}\left[m(\boldsymbol{x}^{j'})\right] + \mathbb{E}\left[(m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'}))^2\right] \\
&= \mathbb{E}\left[(m(\boldsymbol{x}) - m(\boldsymbol{x}^{j'}))^2\right] \\
&= \mathbb{E}\left[(m_j(x^{(j)}) - m_j(x^{(j')}))^2\right] = 2\mathbb{V}\left[m_j(x^{(j)})\right]
\end{aligned}$$

where the last line comes from the additive structure of $m(x)$ and the independence of $x^{(j)}$ and $x^{(j')}$.

By Assumption 1 of 2.1, we have a multivariate normal model. Thus, the conditional mean $m(\boldsymbol{x}) = \mathbb{E}\left[y|\boldsymbol{x}\right]$ will be a linear function of $\boldsymbol{x}$. We now write $m(\boldsymbol{x}) = \sum_{i=1}^{p} \beta_i x^{(i)}$. Using this with the MDA expression from above we have:

$$\text{MDA}(x^{(j)}) = 2\beta_j^2 \mathbb{V}\left[x^{(j)}\right]$$

We will now use the covariance between y and $\boldsymbol{x}^j$ to find an expression for β:

$$\begin{aligned}
\tau_j = \mathbb{E}\left[yx^{(j)}\right] &= \mathbb{E}\left[x^{(j)}\mathbb{E}\left[y|\boldsymbol{x}\right]\right] = \mathbb{E}\left[x^{(j)}\mathbb{E}\left[\sum_{i=1}^{p}\beta_i x^{(i)} + \epsilon|\boldsymbol{x}\right]\right] \\
&= \beta_1\mathbb{E}\left[x^{(1)}x^{(j)}\right] + \ldots + \beta_j\mathbb{E}\left[(x^{(j)})^2\right] + \ldots + \beta_p\mathbb{E}\left[x^{(p)}x^{(j)}\right] \\
&= [\beta C]_j
\end{aligned}$$

so, $\beta_j = [C^{-1}\tau]_j$ which exists because of the invertibility of the covariance matrix C.
To find the inverse of the covariance matrix, first let $C^{-1} = aI_p + b\mathbb{1}\mathbb{1}^\top$ and solve for a and b.

$$\begin{aligned}
CC^{-1} &= ((1-c)I_p + c\mathbb{1}\mathbb{1}^\top)(aI_p + b\mathbb{1}\mathbb{1}^\top) \\
&= a(1-c)I_p + ac\mathbb{1}\mathbb{1}^\top + b(1-c)\mathbb{1}\mathbb{1}^\top + pbc\mathbb{1}\mathbb{1}^\top \\
&= a(1-c)I_p + (ac + b(a-c)pbc)\mathbb{1}\mathbb{1}^\top
\end{aligned}$$

This is equal to an identity matrix if:

$$\begin{cases} a = \frac{1}{1-c} \\ b = \frac{-c}{(1-c)(1-c+pc)} \end{cases}$$

Now, using Assumption 3 of 2.1, we can write the MDA as follows. Noting that $\mathbb{V}\left[x^{(j)}\right] = 1$,

$$\begin{aligned}
\text{MDA}(x^{(j)}) &= 2([C^{-1}\tau]_j)^2\mathbb{V}\left[x^{(j)}\right] = 2([C^{-1}]_{jk}(p-1)\tau_0 + [C^{-1}]_{jj}\tau_0)^2 \\
&= 2(b(p-1)\tau_0 + (a+b)\tau_0)^2 \\
&= 2\left(\tau_0\left(\frac{-c(p-1)}{(1-c)(1-c+pc)} + \frac{1}{1-c} + \frac{-c}{(1-c)(1-c+pc)}\right)\right)^2 = 2\left(\frac{\tau_0}{1-c+pc}\right)^2
\end{aligned}$$

$\diamond$

2.2 Empirical MDA

In practice, when we do not have access to $m(\boldsymbol{x})$ we must use an estimator and an estimated prediction error $\hat{R}$. As described above, the RF estimator is $m_{M,n}(\boldsymbol{x}; \Theta_1, \ldots, \Theta_M, D_n)$. Over a sample $D_n = \{(\boldsymbol{x}_i, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ the estimated prediction error of this estimator is:

$$\hat{R}(m_{M,n}, D_n) = \frac{1}{|D_n|} \sum_{(\boldsymbol{x}_i, y_i) \in D_n} (y_i - m_{M,n}(\boldsymbol{x}_i; \Theta, D_n))^2$$

In this scenario, we can define the empirical MDA of a feature $x^{(j)}$ in a RF of M trees as follows.

$$\widehat{MDA}(x^{(j)}) = \frac{1}{M} \sum_{l=1}^{M} \left[\hat{R}\left(m_{M,n}(\cdot; \Theta_l), D_{l,n}^j \right) - \hat{R}\left(m_{M,n}(\cdot; \Theta_l), D_{l,n} \right) \right] \tag{3}$$

where $D_{l,n}$ is the OOB dataset for the lth tree and $D_{l,n}^j$ is the OOB dataset with feature $x^{(j)}$ permuted [3].

3 Comparison of Theoretical and Empirical MDA

Using data defined to satisfy Assumption 2.1, we will compare the theoretical and empirical MDA metric. The theoretical metric can be plotted using results from Theorem 2.1. To measure an empirical MDA, we conducted simulations using the well-established scikit-learn package for the `RandomForestRegressor` and the `permutation_importance()` function to measure the MDA.

3.1 Simulation Setting

Upon further analysis of the proof of Theorem 2.1 outlined in Gregorutti et. al. [3], we found that the author failed to specify the necessary bounds on τ_0 and c. Specifically, when the data is defined as in (3), the constructed covariance matrix is only valid under certain conditions. The covariance matrix must be positive semi-definite, meaning it must have non-negative eigenvalues. We look at the examples illustrated in their proof where the different numbers of features, p, range from 1 to 5. The eigenvalues of the covariance matrix for each p are defined as follows:

$$
\begin{aligned}
p = 1: &\ \lambda = -\tau_0 + 1 \\
p = 2: &\ \lambda = \frac{c - (c^2 + 8\tau_0^2)^{1/2} + 2}{2} \\
p = 3: &\ \lambda = c - (c^2 + 3\tau_0^2)^{1/2} + 1 \\
p = 4: &\ \lambda = \frac{3c - (9c^2 + 16\tau_0^2)^{1/2} + 2}{2} \\
p = 5: &\ \lambda = 2c - (4c^2 + 5\tau_0^2)^{1/2} + 1
\end{aligned}
$$

In plotting these equations, we can see that the eigenvalues of the covariance matrix are non-negative under certain conditions on c and τ_0. These restrictions were not imposed in the original definition of the data. Figure 1 outlines the curves upper-bounding the regions where the combinations of τ_0 and c produce a valid covariance matrix. We conducted simulations to compare empirical and theoretical results under two settings. One where these restrictions are violated and another where they are not.

In our first setting, as is described in Gregorutti et al. [3],we let $y \sim \mathcal{N}(0, 1)$, $\tau_0 = 0.9$, and plot the theoretical and empirical MDA versus the correlation coefficient, c, for a varying number of features, p.

In our second setting, we again let $y \sim \mathcal{N}(0, 1)$, but now we set τ_0 to be the largest possible value for the associated c and p values such that the covariance matrix is positive semi-definite. This is according to the equation describing the relationship between τ_0 and c, found by setting the $\lambda = 0$ in each of the eigenvalue equations described for $p = 2, 3, 4, 5$. This is akin to using the maximum τ_0 at each c, that is beneath each curve in Figure 1.

3.2 Simulation Results

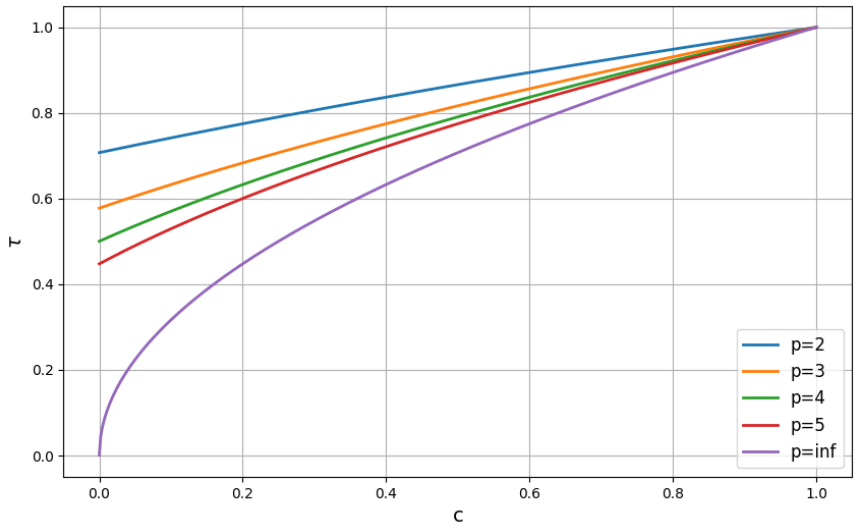


Figure 1: Borders of regions of acceptable τ and c values for $2, 3, 4$ and 5 covariates.

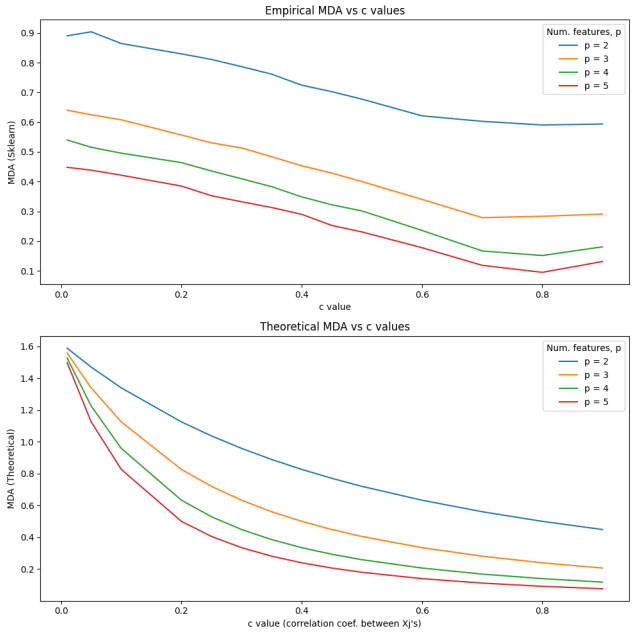


Figure 2: Empirical vs. Theoretical MDA results with fixed $\tau_0 = 0.9$

Comparing Figure 2 and 3 allows us to see that when the restrictions we've found on τ_0, c, and p are met, the empirical results are much closer to the result from Theorem 2.1.

Intuitively, we would expect that the importance of a single feature goes to zero as the number

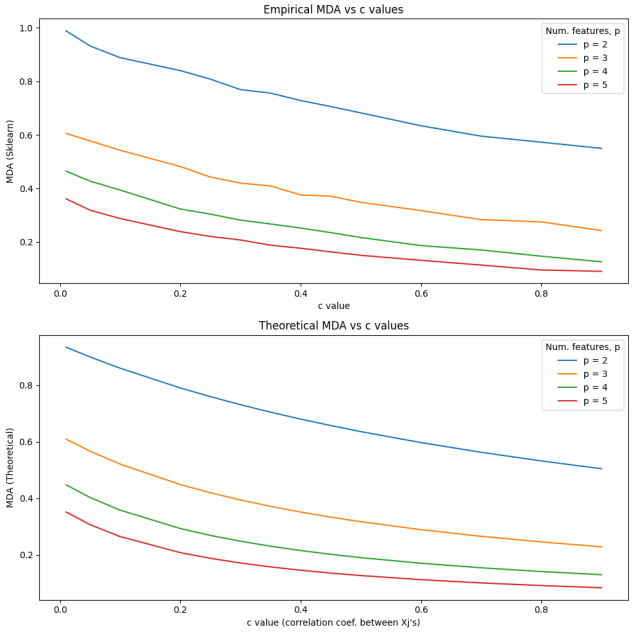


Figure 3: Empirical vs. Theoretical MDA results with maximum valid τ_0

of features, p, gets large. From Theorem 2.1, we can see that the importance of a feature decreases quadratically with p.

In addition to this result, we have found that the region where the covariance matrix is positive semi-definite shrinks as p increases. This tells us that as the number of features increases, fewer values of τ_0 and c satisfy the necessary conditions. Setting λ equal to zero for each of the eigenvalue equations we found in Section 3.1 gives the following relationships between τ_0 and c:

$$p = 2: \ \tau_0 = \sqrt{\frac{1}{2}c + \frac{1}{2}}$$
$$p = 3: \ \tau_0 = \sqrt{\frac{2}{3}c + \frac{1}{3}}$$
$$p = 4: \ \tau_0 = \sqrt{\frac{3}{4}c + \frac{1}{4}}$$
$$p = 5: \ \tau_0 = \sqrt{\frac{4}{5}c + \frac{1}{5}}$$

We hypothesize that the solution to the eigenvalue equation corresponding to the covariance matrix of the same construction with p features is as follows:

$$\tau_0 = \sqrt{(1 - \frac{1}{p})c + \frac{1}{p}}$$

If this is indeed the true relationship, then as the number of features, $p \to \infty$ the curve between τ_0 and c that defines the region where the covariance matrix is positive semi-definite approaches $\tau_0 = \sqrt{c}$. We've include this hypothetical plot on our visualization for comparison. If possible, the proof of this result would be a valuable contribution to the future direction of this work.

We've attached the notebook with the code used to obtain these results at the end of the document

4 Conclusion

In this work, we explored the theoretical and empirical aspects of RFs and their application in assessing variable importance using the MDA metric. When we strengthened previously established assumptions about the covariance matrix's parameter bounds to ensure positive semi-definiteness, simulations demonstrated that empirical results align closely. These findings emphasize the importance of rigorously validating assumptions in statistical models and highlight potential areas for further research, including formal proofs for observed parameter relationships.

Bibliography

[1] Gérard Biau and Erwan Scornet. "A random forest guided tour". In: *TEST* 25.2 (June 2016), pp. 197–227. ISSN: 1863-8260. DOI: 10.1007/s11749-016-0481-7. URL: https://doi.org/10.1007/s11749-016-0481-7.

[2] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[3] Baptiste Gregorutti, Bertrand Michel, and Philippe Saint-Pierre. "Correlation and variable importance in random forests". In: *Statistics and Computing* 27.3 (2017), pp. 659–678. DOI: 10.1007/s11222-016-9646-1. URL: https://doi.org/10.1007/s11222-016-9646-1.

MATH 533 Assignment 3 Code

November 20, 2024

[13]:
```
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.inspection import permutation_importance
import matplotlib.pyplot as plt
```

[14]:
```
def generate_data(n_samples, n_features, c, tau_0):

    #generate the covariance matrix
    I = np.eye(n_features)
    ones = np.ones(n_features)
    C_x = (1-c)*I + c*np.outer(ones, ones)

    #make tau a column vector of tau_0 of lengeth p
    tau = [tau_0]*n_features

    # We need to construct the covariance matrix that they described in their␣
↪theoretical frameowrk
    # This should be a matrix with C (covariance matrxi between Xj's) in the␣
↪top left, tau^T (vector of cov(Xj,Y)) in the bottom left and tau in top␣
↪right, and Var(Y) (1 for std. noraml) in the bottom right

    C = np.block([[C_x, np.array(tau).reshape(-1, 1)], [np.array(tau).
↪reshape(1, -1), 1]])

    #generate noise
    eps = np.random.normal(0, 1)

    #generate X and Y from multivariate normal with mean 0 and the previously␣
↪described covariance matrix
    mean = np.zeros(n_features+1)
    M = np.random.multivariate_normal(mean, C, n_samples)

    #extract X and Y, respectively
    X = M[:, :-1]
    Y = M[:,-1] + eps

    return X, Y
```

```
#Theoretical MDA* result from Gregorutti (2016)
def mda_star(tau_0, c, p):

    theoretical_mda = 2 * (tau_0 / (1 - c + p * c)) ** 2
    return theoretical_mda

#Function to run simulation
def simulation(model, n_samples, p, c, tau_0):

    #get data
    X,Y = generate_data(n_samples, p, c, tau_0)

    # fit off the shelf rf model
    model.fit(X, Y)

    # get empirical MDA values using scikit-learn permutation importance␣
 ↪function (n_repeats specifies how many times average over to shuffle each␣
 ↪feature and measure change in output)
    perm_importance = permutation_importance(model, X, Y, n_repeats=20,␣
 ↪random_state=42)
    importance = perm_importance.importances_mean

    #compute the theoretical mda
    theoretical_mda_val = mda_star(tau_0, c, p)

    # store the results and corresponding p and c values
    sklearn_mda[(p, c)] = np.mean(importance)
    theoretical_mda[(p, c)] = theoretical_mda_val

    return sklearn_mda, theoretical_mda

#plotting function that takes results stored in two dictionaries
def plot_mda_results(sklearn_mdas, theoretical_mdas):
  # initialize
  fig, axes = plt.subplots(2, 1, figsize=(10, 10))

  # plot sklearn_mdas
  for p in p_values:
      # extract the c values and corresponding MDA values for each p
      c_vals = [c for c in c_values]
      mda_vals = [sklearn_mdas[(p, c)] for c in c_vals]

      axes[0].plot(c_vals, mda_vals, label=f"p = {p}")

  axes[0].set_title("Empirical MDA vs c values")
```

```
axes[0].set_xlabel("c value")
axes[0].set_ylabel("MDA (Sklearn)")
axes[0].legend(title='Num. features, p')

# plot theoretical_mdas
for p in p_values:
    # extract the c values and corresponding theoretical MDA values for each p
    c_vals = [c for c in c_values]
    mda_vals = [theoretical_mdas[(p, c)] for c in c_vals]

    axes[1].plot(c_vals, mda_vals, label=f"p = {p}")

axes[1].set_title("Theoretical MDA vs c values")
axes[1].set_xlabel("c value (correlation coef. between Xj's)")
axes[1].set_ylabel("MDA (Theoretical)")
axes[1].legend(title='Num. features, p')

plt.tight_layout()
plt.show()
```

Now we want to run simulations to compare the emperical MDA results with the theoretical derived from the result in Gregorutti et. al.

We compare the emperical results to the theoretical values established in Gregorutti et. al. in two settings.

Setting 1: Fixed tau_0=0.9 for all p and c values (as they did in the paper).

Setting 2 Using the largest tau_0 for each p and c. i.e. following the boundary on the region of invertibility between tau_0 and c for each p, (minus a small value for numerical reasons to stay beneath the curve)derived from solving the eigenvalue equations.

[15]:
```
# set parameters that are constant across settings

model = RandomForestRegressor(n_estimators=200, max_depth=None,␣
↪max_features='sqrt', n_jobs=-1)
p_values = [2,3,4,5]
c_values = [0.01, 0.05, .1, .2, .25, .3, .35, .4, .45, .5, .6, .7, .8, .9]
n_samples = 5000
```

0.1 Setting 1: fixed, invalid tau_0

[16]:
```
#initialize dictionaries to store results
sklearn_mda = {}
theoretical_mda = {}

for p in p_values:
    for c in c_values:
        tau_0 = 0.9
```

```
        #generate the covariance matrix
        C_x = (1-c)*np.eye(p) + c*np.ones((p,p))
        tau = np.array([tau_0] * p).reshape(-1, 1)
        C = np.block([[C_x, tau], [tau.T, 1]])

        sklearn_mdas, theoretical_mdas, = simulation(model, n_samples, p, c,␣
↪tau_0)

# notice here we get warnings for covariance matrix not positive semi-definite
```

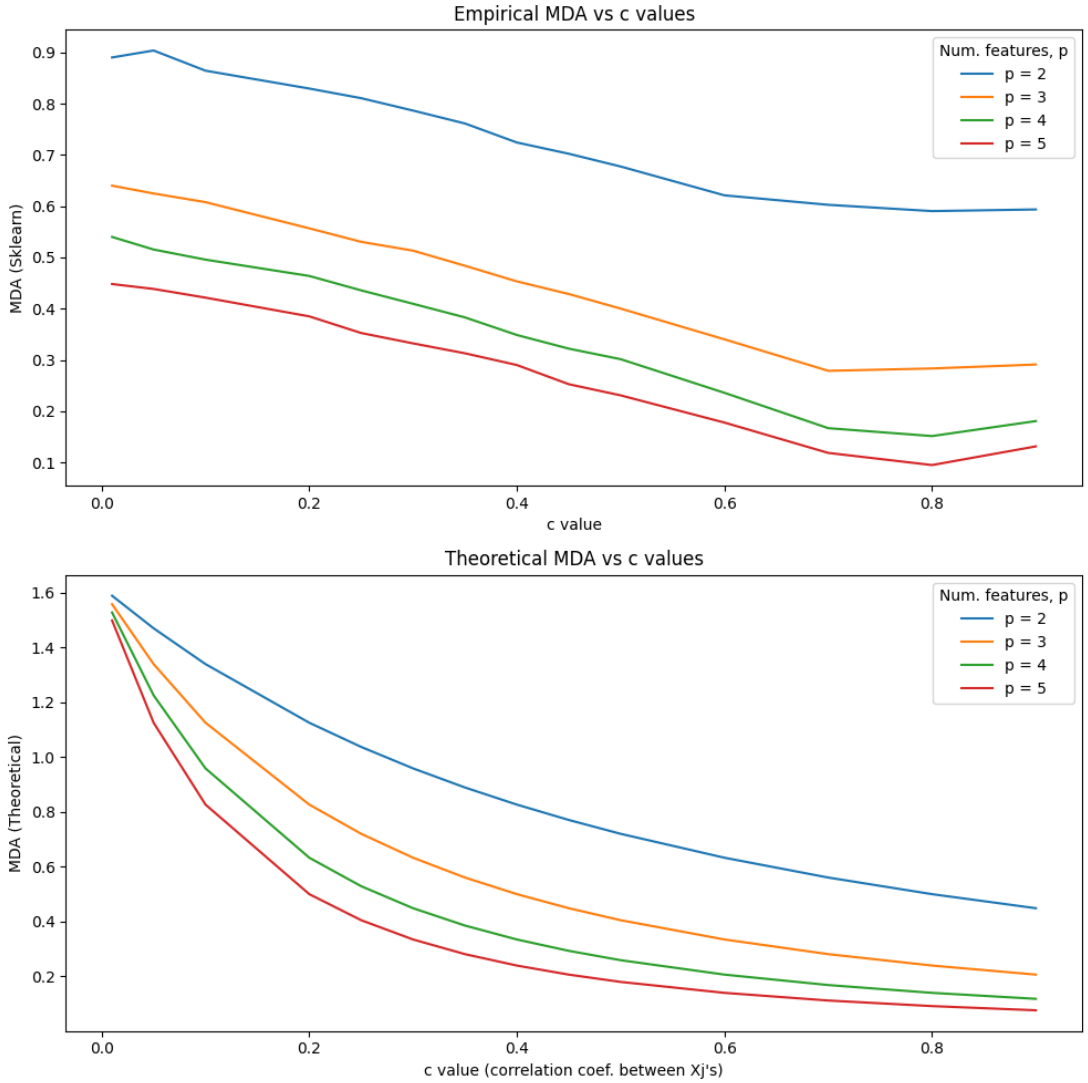
```
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
```

```
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
```

```
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
```

```
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
<ipython-input-14-baef430055b3>:21: RuntimeWarning: covariance is not symmetric
positive-semidefinite.
  M = np.random.multivariate_normal(mean, C, n_samples)
```

[17]:
```
plot_mda_results(sklearn_mdas=sklearn_mda, theoretical_mdas=theoretical_mdas)
```



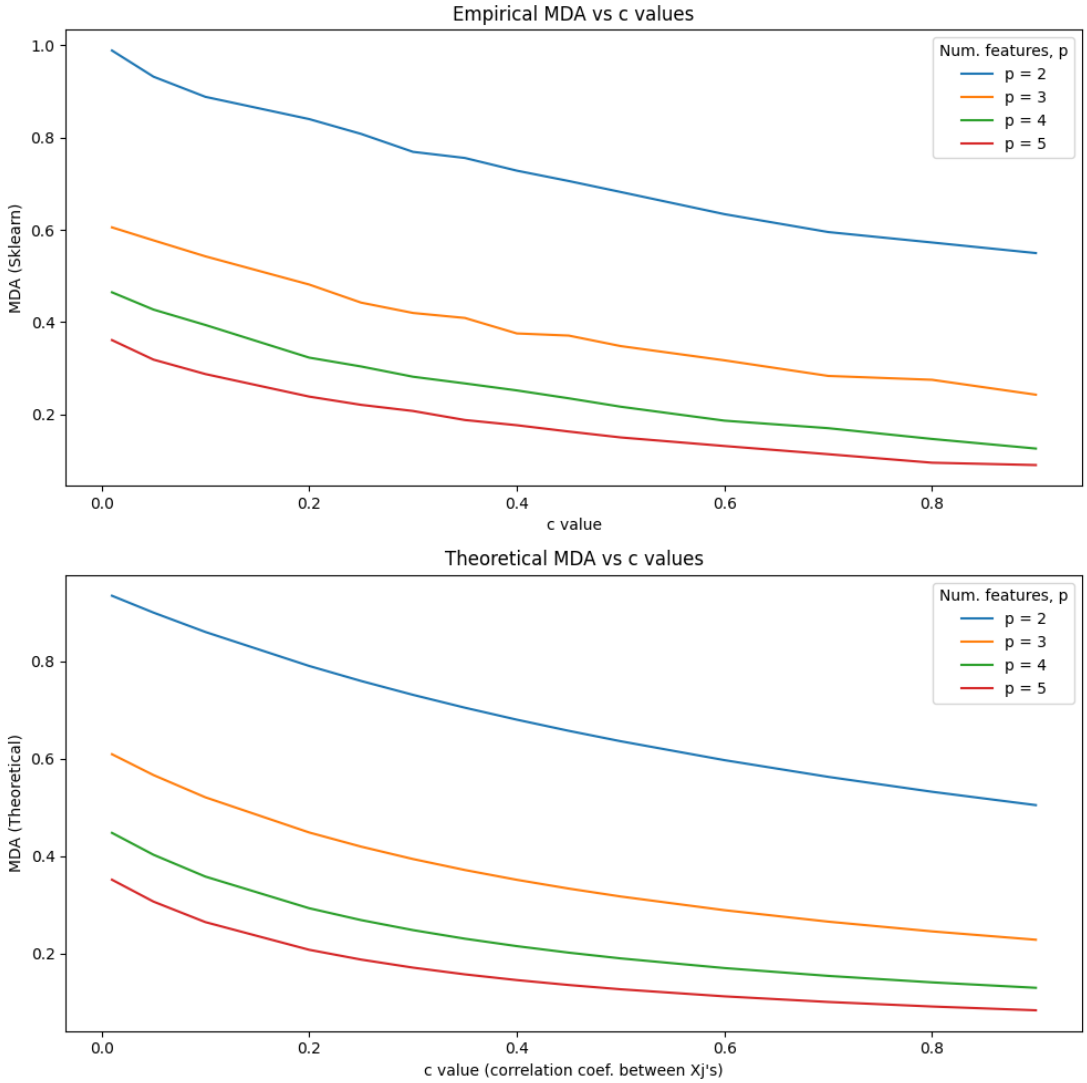
0.2 Setting 2: Maximum valid tau_0 for each c, and p

[18]:
```
#initialize dictionaries to store results
sklearn_mda = {}
theoretical_mda = {}

#define the functions to get the maximum tau_0 for a given c and p
def get_tau_0(c,p):
  num_safety_margin = 0.02
  if p == 2:
    return np.sqrt(1/2 * c +1/2) - num_safety_margin
  elif p == 3:
    return np.sqrt(2/3 * c + 1/3) - num_safety_margin
  elif p == 4:
    return np.sqrt(3/4 * c + 1/4) - num_safety_margin
  else:
    return np.sqrt(4/5 * c + 1/5) - num_safety_margin

for p in p_values:
    for c in c_values:
        tau_0 = get_tau_0(c,p)
        sklearn_mdas, theoretical_mdas, = simulation(model, n_samples, p, c,␣
 ↪tau_0)
```

[19]:
```
plot_mda_results(sklearn_mda, theoretical_mda)
```



We can see clearly here that when we use a fixed τ_0, the emperical results do not align with the theoretical because, as we've said, this is an invalid combination of parameters.

Addiionally, we can see that when we use the largest valid τ_0, (we move along the boundary formed between τ_0 and c for each c and p, our emperical results align well with the theoretical expectation.

[20]:
```
c = np.linspace(0, 1, 500)

# Calculate tau for different values of p
tau_p2 = np.sqrt(1/2 * c + 1/2)
tau_p3 = np.sqrt(2/3 * c + 1/3)
tau_p4 = np.sqrt(3/4 * c + 1/4)
tau_p5 = np.sqrt(4/5 * c + 1/5)
tau_pinf = np.sqrt(c)
```

```
# Plot the curves
plt.figure(figsize=(10, 6))
plt.plot(c, tau_p2, label='p=2', linewidth=2)
plt.plot(c, tau_p3, label='p=3', linewidth=2)
plt.plot(c, tau_p4, label='p=4', linewidth=2)
plt.plot(c, tau_p5, label='p=5', linewidth=2)
plt.plot(c, tau_pinf, label='p=inf', linewidth=2)

# Add labels, legend, and grid
plt.xlabel('c', fontsize=14)
plt.ylabel(r'$\tau$', fontsize=14)
#plt.title('Plots of '+$\tau_0$+' for different values of $p$', fontsize=16)
plt.legend(fontsize=12)
plt.grid(True)

# Show the plot
plt.show()
```