Characterizing Optimal Uncertainty Sets for Linear Regression

Charlie W. Chimento
Technology and Policy Program
Massachusetts Institute of Technology
cchiment@mit.edu

Desiree Waugh
MBAn
Massachusetts Institute of Technology
dwaugh@mit.edu

Abstract

Robust optimization adheres to a set-based, deterministic approach to account for uncertainty, whereas stochastic optimization assumes some underlying probability distribution. A critical component of robust optimization, then, is designing uncertainty sets that formalize a priori knowledge. In this work, we seek to understand what characteristics of a dataset render it amenable to specific uncertainty sets for linear regression. This included building features for 156 datasets, and using an Optimal Classification Tree (OCT) to predict the optimal uncertainty set. Randomness of the data set split caused significant variation in the features that the OCT split on, so multinomial logistic regression was implemented using features frequently selected by the OCT.

1 Problem Setup

At the heart of machine learning is generalizing to unseen data. Robust optimization attempts this by modeling random variables as uncertain parameters belonging to a convex uncertainty set. This approach achieves "robustness" by immunizing against adversarial perturbations in the data. In the case of linear regression, this problem is recognized in the general form

$$\min_{\beta} \max_{\Delta \in U} g(\mathbf{y} - (\mathbf{X} + \mathbf{\Delta})\beta), \tag{1}$$

where $\mathbf{y} \in \mathbb{R}^n$ and $\mathbf{X} \in \mathbb{R}^{nxp}$ with g being some choice of a norm. The uncertainty set, $U \subseteq \mathbb{R}^{nxp}$, formalizes the practitioner's assumptions about the uncertainty in the data matrix. The inner maximization problem reflects the emphasis on training against the worst possible errors, as the permutation matrix Δ is chosen to maximize what the outer problem seeks to minimize–hence adversarial.

1.1 Brief History

Early attempts to account for uncertainty set parameters were to find those equal to the worst-case value. While this approach achieves robustness, it comes at a cost of being overly conservative. Ben-Tal, Nemirovski, and El-Ghaoui addressed this by bounding parameters to ellipsoidal uncertainty sets [1]. This is unfavorable from a computational perspective, however, because a robust framework with ellipsoidal uncertainty sets transforms linear programs to second-order cone problems [5]. Other work improved this by developing a robust optimization approach with polyhedral uncertainty sets, thus retaining the linearity of the constraints [5]. Successful approaches in maintaining the linearity of programs in their robust formulation leverage strong duality. Another foray of robust optimization that has gained attention is consideration of dynamic decision-making. In this setting, parameters are realized over time, and decisions can be made in a multi-stage fashion. This project, however, focuses on static robust optimization.

MIT 9.520/6.860: Statistical Learning Theory and Applications (Fall 2016).

1.2 Uncertainty Sets

The space of possible uncertainty sets is bounded by some matrix norm, and in the case of singular values, a standard norm. Norms that are often used are Frobenius, Induced, and p-Spectral.

$$U_{F(q)} = \{\Delta \in \mathbb{R}^{nxp} : ||\Delta||_{F-q} \leq \rho\} \tag{2}$$
$$U_{(r,q)} = \{\Delta \in \mathbb{R}^{nxp} : ||\Delta||_{r,q} \leq \rho\} \tag{3}$$
$$U_{\sigma_p} = \{\Delta \in \mathbb{R}^{nxp} : ||\mu(\Delta)||_p \leq \rho\} \tag{4}$$

For the Frobenius norm, $||\Delta||_{F-q} := \left(\sum_{i,j} |\Delta_{ij}|^q\right)^{1/q}$, for the induced norm $||\Delta||_{r,q} := \max_x \frac{||\Delta x||_q}{||x||_r}$, and the Schatten norm is a vector p-norm on the eigenvalues of the matrix, $\mu(\Delta)$ [2].

1.3 Choosing an Uncertainty Set

Variations of these uncertainty sets render flexibility to implicitly design a model to be robust against particular kinds of perturbations. Whereas the Frobenius matrix norm implies unstructured uncertainty throughout the data matrix, the uncertainty set constrained by $U_{(1,2)}$, is a reformulation of Lasso [1]. Given a data set, the question of interest becomes, *how do I choose my uncertainty set*? In this work we judge the performance with Least Absolute Deviations on the downstream task of linear regression.

2 Importance

At the heart of machine learning is a balance of enforcing a priori intuition against randomness. Robust optimization achieves this with uncertainty sets. Thus, it is important for decision makers to understand what the uncertainty set implicitly formalizes and its connection to other regularization techniques. A useful example of this is found in Artzner et al, which provided a set of axioms that gives explicit control to incorporate risk preferences by formalizing a class of coherent risk measures [8].

In this field, a plethora of efforts have worked to propose approaches tailored to specific settings. As briefly surveyed above, these settings include single-stage and multi-stage decisions, and settings with uncertainty in parameter estimation with the parameter as either static or stochastic. The variety of approaches provides flexibility to the decision maker to pick the robust optimization formulation that properly locates uncertainty and is tractable. These variations trace an insistence to leverage historical observations of random variables as direct inputs.

This work focuses on the static, linear regression problem with uncertainty in the data. It seeks to provide an empirical foothold to extend theory by suggesting how to interpret characteristics of a dataset in order to predict the optimal uncertainty set for linear regression.

3 Data

The meta-dataset was created from built-in R datasets using the RDatasets package in Julia. Datasets with categorical variables were excluded to avoid high-dimensionality inherent with using binary dummy variables for categorical entries. Also excluded were datasets with fewer than two features (to enable calculation of correlation between different features), and datasets with fewer than 40 observations. The R datasets spanned a variety of domains including economics, health, and biological systems.

For the purposes of optimizing over a particular uncertainty set to find β, the last column in the R dataset was treated as the independent variable. For this research problem, it did not matter which specific column was treated as the independent variable, because the goal was to find the optimal uncertainty set for any given linear regression problem. Missing data was imputed using Optimal k-NN imputation [4], which uses a formulation blind to downstream tasks.

4 Methods

If the goal is to pick the uncertainty set to minimize absolute deviations of linear regression given a data set, a dataset must be constructed with appropriate labels. The process involved four steps:

1. Develop code to find the optimal uncertainty set (of a specific type, such as $U_{(1,2)}$) for a given dataset
2. Develop code to find which specific type of uncertainty set (i.e. $U_{(1,2)}$, $U_{(2,\infty)}$, etc.) minimizes absolute deviations in linear regression
3. Generate features to be calculated for each dataset
4. Evaluate the performance of an OCT in predicting the optimal type of uncertainty set, given the generated features

4.1 Finding the optimal uncertainty set

The metric to be minimized was Mean Absolute Error, so as not to bias towards what has been shown to be equivalent regularization formulations. For example, in [2], Bertismas and Copenhaver showed that

$$\min_{\beta} \max_{\Delta \in U_{(q,p)}} ||\mathbf{y} - (\mathbf{X} + \Delta)\beta||_p = \min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||_p + \lambda||\beta||_q \tag{5}$$

$$\min_{\beta} \max_{\Delta \in U_{\sigma_p}} ||\mathbf{y} - (\mathbf{X} + \Delta)\beta||_2 = \min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||_2 + \lambda||\beta||_2 \tag{6}$$

$$\min_{\beta} \max_{\Delta \in U_{F(p)}} ||\mathbf{y} - (\mathbf{X} + \Delta)\beta||_p = \min_{\beta} ||\mathbf{y} - \mathbf{X}\beta||_p + \lambda||\beta||_{p*} \tag{7}$$

where $p \in [1, \infty)$ satisfies $\frac{1}{p} + \frac{1}{p*} = 1$ for $p* \in [1, \infty]$ for norm duality with the Frobenius uncertainty matrix. These equivalent formulations were leveraged because the right-hand side is readily solvable via JuMP.

Seven uncertainty sets were selected for the classification problem.

- $||\Delta||_{F-1}$
- $||\Delta||_{F-2} = ||\Delta||_{\sigma_2}$
- $||\Delta||_{F-\infty}$
- $||\Delta||_{(1,2)}$
- $||\Delta||_{(2,1)}$
- $||\Delta||_{(2,\infty)}$
- $||\Delta||_{(\infty,2)}$

The right hand side of the expressions in (5,6,7) are amenable to optimization via JuMP. The λ hyperparameter was chosen per performance on a validation set, with the same range of λ values used across all uncertainty sets.

In this fashion, the labels for an optimal uncertainty set were assigned according to the uncertainty set that achieved the lowest MAE.

4.2 Feature Engineering

Once a method was established to identify the optimal uncertainty set, features were generated for each dataset. Chosen features were

- R^2 for $\hat{\beta}$ coming from Ridge Regression
- Percentage of missing data points in $\mathbf{y}$
- Percentage of missing data points in $\mathbf{X}$
- Number of observations

- Number of features
- Percentage of outliers in $\mathbf{y}$
- Mean Pairwise Correlation (transformed mean of Fischer's Z-statistic)
- Mean normalized variance of $\mathbf{X}$
- Median normalized variance of $\mathbf{X}$
- Range of normalized variance in $\mathbf{X}$
- Percentage of features that are correlated above a certain threshold in $\mathbf{X}$
- Variance in $\mathbf{y}$ divided by mean in $\mathbf{y}$

4.3 Feature discussion and justification

If a primary benefit of OCTs over Random Forests and Boosted Trees are their interpretability, then it is important to understand what each generated feature represents.

The intuition of using R^2 was to proxy how amenable a dataset is to a linear model. But calculating R^2 involves $\hat{\mathbf{y}}$, and that necessitates a $\hat{\beta}$. Statistical learning theory shows us that different approaches to estimating β carry different biases. For instance, using OLS biases a solution towards principal components. The normal equations, $\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, yield the $\hat{\beta}$ that minimizes empirical squared loss, but if $d >> n$, taking the inverse of $\mathbf{X}^T\mathbf{X}$ becomes less computationally feasible. Because the datasets ranged from $n >> d$ to $d << n$, it was also not feasible to use $\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ (1) if $n > d$, and $\beta = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}$ (2) if $n < d$. While (1) minimizes squared loss, (2) minimizes the squared sum of the components of s.t. $Y = X$. So the R^2 feature would vary depending on whether $n > d$ or $n < d$.

For these reasons, ridge regression was chosen because it would scale well across variable sizes of n and d using the Gurobi solver through JuMP. The ρ was selected (from a consistent range) that minimized MSE on a hold-out validation set.

Intuitively, the effect of missing data points on MAE would be more drastic if observations with missing features were discarded rather than imputed via kNN, but the data was imputed so as to conserve the number of available datasets with continuous-valued features.

4.4 Optimal Classification Tree

Interpretability is key in this effort, because the goal is to understand what characteristics of a dataset might lend themselves to predicting the optimal uncertainty set. Optimal Classification Trees (OCTs) provide interpretable diagrams of which variables the tree splits on. Like Decision Trees, Random Forests, and Boosted Trees, OCTs are nonlinear classifiers. Decision Trees have the downside of being greedy, which algorithms help account for with post-pruning. Random Forests leverage the principle of aggregated wisdom but are similarly unintepretable because they make decisions per averaging the vote of an ensemble of trees. Thus Optimal Classification Trees were the prime candidate for predicting the optimal uncertainty set for linear regression given a dataset.

4.5 Summary of Method

- Find the uncertainty set that minimizes the mean absolute error for linear regression on 156 datasets pulled from the R-repository
- Generate features for each dataset (mean pairwise correlation, etc...)
- Train an Optimal Classification Tree and evaluate its performance in predicting the best uncertainty set

5 Results

Initial results showed performance on-par or worse than a baseline predictor that chooses the most common label in the training set. In order to improve performance, the number of possible classifications was reduced to the three most commonly selected ($U_{(2,1)}$, $U_{(2,\infty)}$, $U_{(\infty,2)}$). The variables that the OCT split on changed due to randomness in the data split, so a multinomial logistic regression

model was built to predict the three most common classes using three variables highlighted by the OCT. A geometric comparison of the feasible set for β as inferred by the uncertainty set equivalent formulations (5,6,7) elucidates the nature of the solution for the uncertain parameter, β , a given uncertainty set biases towards.

5.1 Optimal Classification Tree Behavior

Optimal Classification Trees, although they have the potential for hyperplane splits, are more interpretable when each split is performed on a single variable. Choosing *max depth* and *min bucket size* with cross-validation, the trained tree tended to return with a selected depth of 2. At a smaller depth, the selection of variables that the OCT decides to split on is telling. However, the selection of variables that the OCT split on were very sensitive to the data split. The figure below shows examples of three different depth-2 trees.

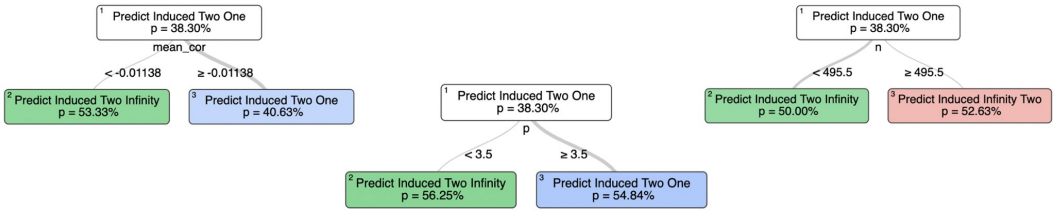


Figure 1: Optimal Classification Trees splitting on different dataset features to predict the optimal uncertainty set

5.2 OCT Performance

Using all 12 of the generated features, the out-of-sample performance was slightly worse than the baseline in the case of choosing among 3 uncertainty sets. When choosing among all 7 constructed uncertainty sets, the performance of the OCT was also worse than the baseline model. Table 1 reports the performance.

Table 1: OCT Prediction Performance as Measured by Mean Accuracy and Accuracy Variance Between Trials

trials = 100	features = all				
# Labels	In-Sample Acc.	Test Acc.	In-Sample Acc. Var.	Test Acc. Var.	Baseline Acc.
3	58.10%	40.79%	0.46%	0.61%	42.30%
7	41.51%	23.83%	0.59%	0.36%	27.20%

5.3 Further Results with Logistic Regression

While the OCT did not yield strong performance, it did consistently highlight a few variables as important in determining the optimal uncertainty set. Thus it seemed a natural extension to explore the accuracy of a multinomial logistic regression model regressing on these highlighted features. The results beat the baseline out-of-sample accuracy. For comparison, a logistic regression model was trained and tested using all the generated features and the mean performance was 32.80% accuracy, suggesting that the OCT served a purpose in highlighting the generated features which are indicative of which uncertainty set would be optimal for minimizing MAE on linear regression. Table 2 summarizes the performance of the logistic regression using the features identified by the OCT.

Table 2: Logistic Regression Prediction Performance as Measured by Mean Accuracy and Accuracy Variance Between Trials

trials = 100 features = n, p, mean pairwise corr.					
# Labels	In-Sample Acc.	Test Acc.	In-Sample Acc. Var.	Test Acc. Var.	Baseline Acc.
3	49.83%	42.97%	0.10%	0.41%	42.30%

5.4 Geometric Interpretation

The uncertainty sets that most often returned the best out-of-sample performance were $U_{(1,2)}$, $U_{(2,\infty)}$, and $U_{(\infty,2)}$. This translates to an l_1, l_2 and l_∞ norm on β, respectively. A geometric perspective lends insight on the type of solutions for β that each of these uncertainty sets biases towards. The figure below visualizes the shape of the feasible region for two-dimensions of β. Note that the optimal solution will lie at the corners for l_1 and l_2, or in the case of the circle, anywhere along the circle border.

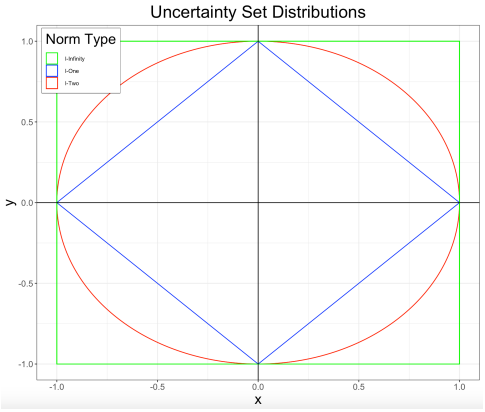


Figure 2

The geometry demonstrates the bias of different norms on β towards certain solutions, each with a distinct nature. $U_{(2,\infty)}$ will almost certainly not be sparse in β and $U_{(\infty,2)}$ uncertainty sets will tend to produce β without bias towards either sparsity or density.

A natural question, then, is *what characteristics of a dataset would suggest that sparsity as opposed to just minimizing each component of β would be preferable*? If the simulation were to be run again, it would be interesting to generate features characterizing a dataset that are known to be more amenable to sparse as opposed to dense βs and see whether the OCT uses this feature in classifying the equivalent optimal uncertainty set.

6 Conclusions

Identifying the optimal uncertainty set is an inherently challenging problem because there are many degrees of freedom, from selecting the performance parameter, to dealing with missing values, to randomness in the split for training, validation, and testing.

Further areas of exploration include uncertainty tests built on hypothesis tests. Previous research has shown a way to build uncertainty sets for continuous, independent features using the Kolmogorov-Smirnov test with probabilistic guarantees [3].

Additionally, it was difficult to build a strong machine learning model using only 156 observations. It would be instructive to repeat the procedure with a much larger dataset compiled from a variety of sources and domains.

7 Contributions

Charlie Chimento

- Wrote final paper
- Research on theory for final paper
- Code for feature engineering

Desiree Waugh

- Code to optimize over uncertainty sets and find the optimal uncertainty set for a given dataset
- Code to read in R Datasets and create meta-dataset with 156 observations
- Code for OCT and multinomial logistic regression

References

[1] Ben-Tal, A. & Goryashko, A. & Guslitzer, E. & Nemirovski, A. (2003) Adjustable Robust Solutions of Uncertain Linear Programs, *Springer-Verlag*, Math. Proram. Ser. A 99:351-376.

[2] Bertsimas, D. & Copenhaver, M. (2018) Characterization of the Equivalence of Robustification and Regularization in Linear and Matrix Regression, *European Journal of Operational Research*, pp. 931–942. vol. 270, no.3.

[3] Bertsimas, D. & Gupta, V. & Kallus, N. (2017) Data Driven Robust Optimization, *Berlin Heidelberg and Optimization Society*, pp. 235–292. vol. 167, no.2.

[4] Bertsimas, D. & Dunn, J. (2019) *Machine Learning under a Modern Optimization Lens* Dynamic Ideas.

[5] Bertsimas, D. & Thiele, A. Robust and Data-Driven Optimization: Modern Decision-Making Under Uncertainty (2006).

[6]Ghaoui, Laurent. & Robust Solutions to Least-Squares Problems with Uncertain Data (1997), *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 4, pp. 1035-1064.

[7] Tulabandhula, Theja. & Rudin, Cynthia (2014) Robust Optimization using Machine Learning for Uncertainty Sets.

[8] Artzner, P. & Delbaen, F. (1999) Coherent Measures of Risk. *Mathematical Finance*, 9:203-228.

In [1]:
```
cd("/Users/desireewaugh/Dropbox (MIT)/15.095 Final Project")
```

In [2]:
```
using DataFrames, Plots, StatsBase, JuMP, Gurobi, DataFrames, CSV, Linea
rAlgebra, Random
```

Read in sample dataset from HW1 to test functions

In [55]:
```
# Import data and convert to matrices
X_var = CSV.read("sparseX2.csv", header=false)
Y_var = CSV.read("sparseY2.csv", header=false)

X_matrix = convert(Matrix, X_var)
Y_matrix = convert(Matrix, Y_var)

# Normalize the X_matrix
#X_matrix = zscore(X_matrix); We dont' want to normalize everything toge
ther, because this is effectively cheating by looking at the validation
 and testing values inc alculating a centering factor

test_df = X_var
hcat(X_var, Y_var, makeunique=true)[!,101];
```

Frobenius Norm Sets

In [56]:
```
function frobenius_one(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, abs_beta[1:p])

    @variable(mod, b)
    @variable(mod, abs_Ter1[1:n]>=0)

    @constraint(mod, -beta .<= abs_beta)
    @constraint(mod, beta .<= abs_beta)

    @constraint(mod, [i=1:p], abs_beta[i] <= b) # b = max of abs. value
 of elements of beta, which is the l-infinity norm

    @constraint(mod, -(y - X*beta) .<= abs_Ter1) # abs_Ter1 = abs(y - X*
beta)
    @constraint(mod, (y - X*beta) .<= abs_Ter1)

    @objective(mod, Min, sum(abs_Ter1) + ϱ*b)

    solve(mod)
    return(getvalue(beta))

end
```

Out[56]: frobenius_one (generic function with 1 method)

In [58]:
```
function frobenius_two(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, Ter1>=0)
    @variable(mod, Ter2>=0)

    @constraint(mod, Ter1 >= norm(y - X*beta)) # norm returns the l-2 no
rm
    @constraint(mod, Ter2 >= ϱ*norm(beta))

    @objective(mod, Min, Ter1 + Ter2)

    solve(mod)
    return(getvalue(beta))
end
```

Out[58]: frobenius_two (generic function with 1 method)

In [60]:
```
function frobenius_infinity(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, abs_beta[1:p])
    @variable(mod, max_Ter1)

    @variable(mod, abs_Ter1[1:n]>=0)

    @constraint(mod, -beta .<= abs_beta) # abs_beta is the absolute valu
e of the beta vector
    @constraint(mod, beta .<= abs_beta)

    @constraint(mod, -(y - X*beta) .<= abs_Ter1) # abs_Ter1 = abs(y - X*
beta)
    @constraint(mod, (y - X*beta) .<= abs_Ter1)

    @constraint(mod, [i=1:n], abs_Ter1[i] <= max_Ter1) # max_Ter1 is the
l-infinity norm of the vector (y-X*beta)

    @objective(mod, Min, max_Ter1 + ϱ*sum(abs_beta))

    solve(mod)
    return(getvalue(beta))

end
```

Out[60]: frobenius_infinity (generic function with 1 method)

In [62]:
```
function induced_one_two(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, abs_Ter1[1:n]>=0)
    @variable(mod, Ter2>=0)

    @constraint(mod, -(y - X*beta) .<= abs_Ter1) # abs_Ter1 = abs(y - X*
beta)
    @constraint(mod, (y - X*beta) .<= abs_Ter1)

    @constraint(mod, Ter2 >= ϱ*norm(beta)) # Ter2 is ϱ times the l-2 norm
of beta

    @objective(mod, Min, sum(abs_Ter1) + Ter2)

    solve(mod)
    return(getvalue(beta))
end
```

Out[62]: induced_one_two (generic function with 1 method)

In [64]:
```
function induced_two_one(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, Ter1>=0)
    @variable(mod, abs_beta[1:p])

    @constraint(mod, Ter1 >= norm(y - X*beta)) # Ter1 is the l-2 norm of
(y - X*beta)

    @constraint(mod, -beta .<= abs_beta)
    @constraint(mod, beta .<= abs_beta)

    @objective(mod, Min, Ter1 + ϱ*sum(abs_beta))

    solve(mod)
    return(getvalue(beta))
end
```

Out[64]: induced_two_one (generic function with 1 method)

In [66]:
```
function induced_two_infinity(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, Ter1>=0)
    @variable(mod, abs_beta[1:p])
    @variable(mod, b)

    @constraint(mod, Ter1 >= norm(y - X*beta)) # Ter1 is the l-2 norm of
(y - X*beta)

    @constraint(mod, -beta .<= abs_beta)
    @constraint(mod, beta .<= abs_beta)

    @constraint(mod, [i=1:p], abs_beta[i] <= b) # b = max of abs. value
 of elements of beta, which is the l-infinity norm

    @objective(mod, Min, Ter1 + ϱ*b)

    solve(mod)
    return(getvalue(beta))
end
```

Out[66]: induced_two_infinity (generic function with 1 method)

In [68]:
```
function induced_infinity_two(X, y, ϱ)
    mod = Model(solver=GurobiSolver(OutputFlag=0, Quad=1, PSDTol=1e-3))

    n = size(X)[1]
    p = size(X)[2]

    @variable(mod, beta[1:p])
    @variable(mod, max_Ter1)
    @variable(mod, abs_Ter1[1:n]>=0)
    @variable(mod, Ter2>=0)

    @constraint(mod, -(y - X*beta) .<= abs_Ter1) # abs_Ter1 = abs(y - X*
beta)
    @constraint(mod, (y - X*beta) .<= abs_Ter1)

    @constraint(mod, [i=1:n], abs_Ter1[i] <= max_Ter1) # max_Ter1 is l-i
nfinity norm of (y - X*beta)

    @constraint(mod, Ter2 >= ϱ*norm(beta)) # Ter2 is ϱ times the l-2 norm
of beta

    @objective(mod, Min, max_Ter1 + Ter2)

    solve(mod)
    return(getvalue(beta))

end
```

Out[68]:
```
induced_infinity_two (generic function with 1 method)
```

Function to split a dataset into training, validation, and testing sets

In [70]:
```
function splitter(X_data, Y_data, percent_train, percent_valid)
    # Concatenate X and Y data
    data = hcat(Y_data, X_data)

    # Shuffle rows of X and Y data combined
    data = data[shuffle(1:end), :]

    # Split X and Y data
    Y_shuffled = data[:, end]
    X_shuffled = data[:, 1:end-1]

    # Get indices for splitting your data
    n = size(data)[1]
    train_ind = Int(round(percent_train*n))
    valid_ind = train_ind + Int(round(percent_valid*n))
    test_ind = n

    # Split X data based on indices
    X_train_data = X_shuffled[1:train_ind, :]
    X_valid_data = X_shuffled[train_ind+1:valid_ind, :]
    X_test_data = X_shuffled[valid_ind+1:n, :]

    # Split Y data based on indices
    Y_train_data = Y_shuffled[1:train_ind, :]
    Y_valid_data = Y_shuffled[train_ind+1:valid_ind, :]
    Y_test_data = Y_shuffled[valid_ind+1:n, :]

    #Now we normalize features!

    #Center features
    for i in 1:size(X_train_data,2)
        X_train_data[:,i] = X_train_data[:,i] .- mean(X_train_data[:,i
]);
        X_valid_data[:,i] = X_valid_data[:,i] .- mean(X_train_data[:,i
]);
        X_test_data[:,i] = X_test_data[:,i] .- mean(X_train_data[:,i]);
    end

    #Calculate the std for each feature
    num_dims = size(X_train_data,2)
    std_feat = zeros(num_dims)
    for i in 1:num_dims
      std_feat[i] = std(X_train_data[:,i]);
    end

    #scale the features
    for i in 1:num_dims
        X_train_data[:,i]=X_train_data[:,i] ./std_feat[i]
        X_valid_data[:,i] = X_valid_data[:,i] ./ std_feat[i];
        X_test_data[:,i] = X_test_data[:,i] ./ std_feat[i];
    end

    #Center Labels
    mean_train_y = mean(Y_train_data);
    Y_train_data =  Y_train_data .- mean_train_y;
    Y_valid_data = Y_valid_data .- mean_train_y;
```

```
    Y_test_data = Y_test_data .- mean_train_y;


    return(X_train_data, X_valid_data, X_test_data, Y_train_data, Y_vali
d_data, Y_test_data)

end
```

Out[70]: splitter (generic function with 1 method)

In [71]:
```
# Split the sample dataset so we can test the functions
X_train, X_valid, X_test, Y_train, Y_valid, Y_test = splitter(X_matrix,
Y_matrix, 0.5, 0.25);
```

MAE function to validate ρ values and evaluate uncertainty sets

In [72]:
```
function MAE(X_mat, y_vec, beta_vec)
    return(sum(broadcast(abs, y_vec - X_mat*beta_vec)))
end
```

Out[72]: MAE (generic function with 1 method)

Finds the best value of ρ for a robust optimization problem through validation

In [73]:
```
function row_tester(row_list, X_train, Y_train, X_valid, Y_valid, func)

    MAE_errors = []

    # Loop through different possible values for ϱ
    # Calculate beta value (training data) and then MAE value (validatio
n data) for that particular ϱ
    # Save your MAE values in an array
    for p in row_list
        beta_temp = func(X_train, Y_train, p)
        MAE_tmp = MAE(X_valid, Y_valid, beta_temp)
        append!(MAE_errors, MAE_tmp)
    end

    min_index = argmin(MAE_errors)  # find the minimum MAE
    return(row_list[min_index]) # return the value of ϱ that gave the mi
nimum MAE on the validation set

end
```

Out[73]: row_tester (generic function with 1 method)

Test all uncertainty sets on a dataset and return the name of the one that resulted in the lowest test MAE

In [75]:
```
function best_uncertainty_set(X_train, X_valid, X_test, Y_train, Y_valid
, Y_test, row_range)
    # Set up dictionary with starting values (to be replaced)
    sets_dict = Dict([("Frobenius One", 0.5), ("Frobenius Two", 0.5), (
"Frobenius Infinity", 0.5),
                      ("Induced One Two", 0.5), ("Induced Two One", 0.5),
                        ("Induced Two Infinity", 0.5), ("Induced Infinity Tw
o", 0.5)])

    # Find best ϱ for Frobenius One uncertainty set, and save MAE for Fr
obenius One beta
    row_frob_one = row_tester(row_range, X_train, Y_train, X_valid, Y_va
lid, frobenius_one)
    beta_frob_one = frobenius_one(X_train, Y_train, row_frob_one)
    sets_dict["Frobenius One"] = MAE(X_test, Y_test, beta_frob_one)

    # Find best ϱ for Frobenius Two uncertainty set, and save MAE for Fr
obenius Two beta
    row_frob_two = row_tester(row_range, X_train, Y_train, X_valid, Y_va
lid, frobenius_two)
    beta_frob_two = frobenius_two(X_train, Y_train, row_frob_one)
    sets_dict["Frobenius Two"] = MAE(X_test, Y_test, beta_frob_two)

    # Find best ϱ for Frobenius Infinity uncertainty set, and save MAE f
or Frobenius Infinity beta
    row_frob_infinity = row_tester(row_range, X_train, Y_train, X_valid,
Y_valid, frobenius_infinity)
    beta_frob_infinity = frobenius_infinity(X_train, Y_train, row_frob_i
nfinity)
    sets_dict["Frobenius Infinity"] = MAE(X_test, Y_test, beta_frob_infi
nity)

    # Find best ϱ for Induced One Two uncertainty set, and save MAE for
 Induced One Two beta
    row_induced_12 = row_tester(row_range, X_train, Y_train, X_valid, Y_
valid, induced_one_two)
    beta_induced_12 = induced_one_two(X_train, Y_train, row_induced_12)
    sets_dict["Induced One Two"] = MAE(X_test, Y_test, beta_induced_12)

    # Find best ϱ for Induced Two One uncertainty set, and save MAE for
 Induced Two One beta
    row_induced_21 = row_tester(row_range, X_train, Y_train, X_valid, Y_
valid, induced_two_one)
    beta_induced_21 = induced_two_one(X_train, Y_train, row_induced_21)
    sets_dict["Induced Two One"] = MAE(X_test, Y_test, beta_induced_21)

    # Find best ϱ for Induced Two Infinity uncertainty set, and save MAE
for Induced Two Infinity beta
    row_induced_2inf = row_tester(row_range, X_train, Y_train, X_valid,
Y_valid, induced_two_infinity)
    beta_induced_2inf = induced_two_infinity(X_train, Y_train, row_induc
ed_2inf)
    sets_dict["Induced Two Infinity"] = MAE(X_test, Y_test, beta_induced
_2inf)

    # Find best ϱ for Induced Infinity Two uncertainty set, and save MAE
```

```
    for Induced Infinity Two beta
        row_induced_inf2 = row_tester(row_range, X_train, Y_train, X_valid,
    Y_valid, induced_infinity_two)
        beta_induced_inf2 = induced_infinity_two(X_train, Y_train, row_induc
    ed_inf2)
        sets_dict["Induced Infinity Two"] = MAE(X_test, Y_test, beta_induced
    _inf2)

        # Return name of uncertainty set that had t
        return(findmin(sets_dict)[2])
    end
```

Out[75]: best_uncertainty_set (generic function with 1 method)

In [125]:
```
    function best_uncertainty_reduced(X_train, X_valid, X_test, Y_train, Y_v
    alid, Y_test, row_range)
        # Set up dictionary with starting values (to be replaced)
        sets_dict = Dict([("Induced Two One", 0.5),
                          ("Induced Two Infinity", 0.5),
                ("Induced Infinity Two", 0.5)])

        # Find best ϱ for Induced Two One uncertainty set, and save MAE for
     Induced Two One beta
        row_induced_21 = row_tester(row_range, X_train, Y_train, X_valid, Y_
    valid, induced_two_one)
        beta_induced_21 = induced_two_one(X_train, Y_train, row_induced_21)
        sets_dict["Induced Two One"] = MAE(X_test, Y_test, beta_induced_21)

        # Find best ϱ for Induced Two Infinity uncertainty set, and save MAE
    for Induced Two Infinity beta
        row_induced_2inf = row_tester(row_range, X_train, Y_train, X_valid,
    Y_valid, induced_two_infinity)
        beta_induced_2inf = induced_two_infinity(X_train, Y_train, row_induc
    ed_2inf)
        sets_dict["Induced Two Infinity"] = MAE(X_test, Y_test, beta_induced
    _2inf)

        # Find best ϱ for Induced Infinity Two uncertainty set, and save MAE
    for Induced Infinity Two beta
        row_induced_inf2 = row_tester(row_range, X_train, Y_train, X_valid,
    Y_valid, induced_infinity_two)
        beta_induced_inf2 = induced_infinity_two(X_train, Y_train, row_induc
    ed_inf2)
        sets_dict["Induced Infinity Two"] = MAE(X_test, Y_test, beta_induced
    _inf2)

        # Return name of uncertainty set that had t
        return(findmin(sets_dict)[2])
    end
```

Out[125]: best_uncertainty_reduced (generic function with 1 method)

Build a function that returns features for the OCT to split on

Features: R^2, domain, number of features, number of observations, normalized variance of the labels, avg normalzied variance of the features, median normalized variance of the features, range of the normalized variance of the features, percentage of outliers in the labels, percent of features that have at least some threshold of correlation, percent of missing labels, avg percent of missing values in the features, mean correlation (fisher's z-test).

In [127]:
```
using Statistics
```

In [135]:
```
# Create an empty features dataframe
meta_df = DataFrame(
    Dataset_Name = String[],
    Source = String[],
    y_miss_per = Float64[],
    avg_per_miss_x = Float64[],
    r2_rr = Float64[],
    n = Int64[],
    p = Int64[],
    ind_Disp_y = Float64[],
    x_mean_norm_var = Float64[],
    median_x_norm_var = Float64[],
    range_norm_x_var = Float64[],
    y_outliers_per = Float64[],
    x_corr_per = Float64[],
    mean_cor = Float64[],
    uncertainty_set = String[])
```

Out[135]: 0 rows × 15 columns (omitted printing of 7 columns)

Dataset_Name	Source	y_miss_per	avg_per_miss_x	r2_rr	n	p	ind_Disp_y
String	String	Float64	Float64	Float64	Int64	Int64	Float64

In [136]:
```
# Get list of filtered R Datasets - I filtered out certain R packages an
d datasets with fewer than 40 rows
RDataset_List = CSV.read("RDatasets_Filtered.csv");
```

Loop through the R datasets, generate features, and add optimal uncertainty set label

In [137]:
```
using RDatasets
```

In [138]:
```
function knn_impute(dataset)
    # Convert NaN to missing
    for col in 1:size(dataset, 2)
        replace(dataset[!,col], NaN => missing)
    end

    # Cross validation
    grid = IAI.GridSearch(IAI.ImputationLearner(random_seed=2),
        (method=:opt_knn, knn_k = [5:5:100;]))
    best_mod = IAI.get_learner(grid)

    IAI.fit!(best_mod,dataset)
    data_imputed = IAI.transform(best_mod, dataset)
    return(data_imputed)
end
```

Out[138]:
```
knn_impute (generic function with 1 method)
```

In [ ]:
```
for i in 1:size(RDataset_List, 1)
    name = RDataset_List[i,:Dataset] # name of dataset to read in
    source = RDataset_List[i,:Package] # source of dataset to read in
    print(name, " ", source, "\n")
    tmp_df = dataset(source, name) # read in datasets one at a time


    tmp_features = Any[name, source] # Create a list of features to add
 to meta_df

    # Find missing data points
    y_miss_per = perc_missing_y(tmp_df[!,end])
    avg_per_miss_x = perc_missing_x(convert(Matrix, tmp_df[!,1:end-1]))
    append!(tmp_features, y_miss_per)
    append!(tmp_features, avg_per_miss_x)

    # Get a list of the column types in tmp_df
    col_types = []
    for n in names(tmp_df)
        push!(col_types, typeof(tmp_df[!,n]))
    end

    # Skip datasets with non-numeric values
    non_numeric = false
    for t in col_types
        if occursin("Categorical", string(t))
            non_numeric = true
        elseif occursin("String", string(t))
            non_numeric = true
        end
    end

    if non_numeric == true
        continue
    end

    tmp_df = knn_impute(tmp_df) # Impute any missing data

    print(name, " ", source, " " , i, "\n")

    features(tmp_df)
    append!(tmp_features, features(tmp_df))

    push!(meta_df, tmp_features)
end
```

Save metadata as CSV

In [159]:
```
CSV.write("Final_Metadata_Reduced.csv", meta_df)
```

Out[159]: "Final_Metadata_Reduced.csv"

In [134]:
```
#takes in data set, splits into X, and labels y.
function features(data)
    data = convert(Matrix,data)
    X = data[:,1:end-1]
    y = data[:,end]

    # split data
    X_train, X_valid, X_test, Y_train, Y_valid, Y_test = splitter(X,y,0.
5,0.25)

    r2_rr = R2_RR(X_train, X_valid, X_test, Y_train, Y_valid, Y_test) ##
returns r^2 for ridge regression

    n = size(X,1)
    p = size(X,2)

    if mean(y) == 0 && var(y) == 0
        ind_Disp_y = 0
    else
        ind_Disp_y = var(y)/mean(y)
    end

    x_mean_norm_var, x_norm_var = mean_norm_var(X)

    #mean normalized variance of the features
    median_x_norm_var = median(x_norm_var)

    range_norm_x_var = maximum(x_norm_var)-minimum(x_norm_var)

    y_outliers_per = count(i-> -2*std(y)<i<2*std(y), y)/length(y)

    #mean_cor = mean(cor(X)) this is wrong!
    #percentage of features that are correlated above a certain threshol
d
    x_corr_per = 1- count(i-> (-0.4<i<0.4),cor(X))/(length(cor(X))-p)

    ## vectorize correlation matrix. z transform each element, then aver
age, then reverse transform.
    ## This approach requires some assumptions on the data
    mean_cor = correlation_metric(X,p)

    uncertainty_set = best_uncertainty_reduced(X_train, X_valid, X_test,
Y_train, Y_valid, Y_test, collect(0:0.5:10))

    return [r2_rr, n, p,
        ind_Disp_y, x_mean_norm_var, median_x_norm_var,
        range_norm_x_var, y_outliers_per, x_corr_per, mean_cor, uncertai
nty_set]
end
```

Out[134]:
```
features (generic function with 1 method)
```

In [88]:
```
function perc_missing_x(X)
    x_miss = []
    for j in 1:size(X,2)
        append!(x_miss, count(i -> typeof(i)==Missing, X[:,j])/length(X[
:,j]))
    end
    return mean(x_miss)
end
```

Out[88]: perc_missing_x (generic function with 1 method)

In [89]:
```
function perc_missing_y(Y)
    return (count(i -> typeof(i)==Missing, Y[:,1])/size(Y, 1))
end
```

Out[89]: perc_missing_y (generic function with 1 method)

In [90]:
```
function mean_norm_var(X)
    x_norm_var = []
    for i in 1:size(X,2)
        if mean(X[:,i]) == 0
            append!(x_norm_var, 0)
        else
            append!(x_norm_var, var(X[:,i])/mean(X[:,i]))
        end
    end
    return mean(x_norm_var), x_norm_var #mean normalized variance of the
features
end
```

Out[90]: mean_norm_var (generic function with 1 method)

In [93]:
```
function correlation_metric(X,p)
    vec_cor =[]
    mat_cor = cor(X)
    for i in 1:p
        for j in 1:p
            if j>i
                append!(vec_cor,mat_cor[i,j])
            end
        end
    end
    return tanh(mean(atanh.(vec_cor)))
end
```

Out[93]: correlation_metric (generic function with 1 method)

Ridge regression for calculating the β's to find an estimate $\hat{y}$

In [94]:
```
function L2(train_X,train_y,ϱ)
    m = Model(solver=GurobiSolver(PSDTol=1e-3))

    P = size(train_X)[2]
    n = size(train_X)[1]

    @variable(m, z[1:2])
    @variable(m, β[1:P])


    @constraint(m, z[1] >= sum((train_y[i] - transpose(β)*train_X[i,:])^
2 for i in 1:n))
    @constraint(m, z[2] >= sum(β[i]^2 for i in 1:P))


    @objective(m, Min, z[1] + ϱ*z[2])

    solve(m)

    return getvalue(β)
end
```

Out[94]: L2 (generic function with 1 method)

In [3]:
```
y = CSV.read("C:\\Users\\Charlie\\Documents\\Academic\\Courses\\15.095\\
PSets\\HW1\\data\\sparseY2.csv", header = 0);
 X = CSV.read("C:\\Users\\Charlie\\Documents\\Academic\\Courses\\15.095
\\PSets\\HW1\\data\\sparseX2.csv", header = 0);
 X = convert(Matrix,X);
 y = convert(Matrix,y);
```

Adaption of earlier function that uses MSE instead to judge the validation loss because ridge regression optimimizes on squared error

In [95]:
```
function row_tester_MSE(row_list, X_train, Y_train, X_valid, Y_valid, fu
nc)

    MSE_errors = []
    βs = []

    # Loop through different possible values for ϱ
    # Calculate beta value (training data) and then MAE value (validatio
n data) for that particular ϱ
    # Save your MSE values in an array
    for p in row_list
        beta_temp = func(X_train, Y_train, p)
        push!(βs,beta_temp)
        MSE_tmp = MSE(X_valid, Y_valid, beta_temp)
        append!(MSE_errors, MSE_tmp)
    end

    min_index = argmin(MSE_errors)  # find the index of that minimum MSE
    return(βs[min_index]) # return the value of β that gave the minimum M
SE on the validation set

end
```

Out[95]: row_tester_MSE (generic function with 1 method)

In [96]:
```
function MSE(X_valid,Y_valid,beta_temp)
    return sum((Y_valid[i] - transpose(beta_temp)*X_valid[i,:])^2 for i=
1:size(X_valid,1))
end
```

Out[96]: MSE (generic function with 1 method)

In [98]:
```
function R2_RR(X_train, X_valid, X_test, Y_train, Y_valid, Y_test) #retu
rns a different value for R^2 because of randomization of the splits
    #We want to use all the data, because we only care about finding ϱ t
o generate an unbiased $R^2$
    opt_β = row_tester_MSE([0.1,1,5,10,20],X_train,Y_train,X_valid,Y_val
id,L2);
    y_hat = X_test*opt_β
    mean_y = mean(Y_test)
    y_bar = fill(mean_y,size(Y_test,1))
    return 1 - sum((Y_test .- y_hat).^2)/sum((Y_test .- y_bar).^2)
end
```

Out[98]: R2_RR (generic function with 1 method)

If we are going to use R^2 as a feature, we must understand what it is doing so that we can interpret the OCT results. One possible interpretation is that R^2 guages how amenable a dataset is to a linear model. But to calculate, we need estimates, $\hat{y}$, that come from β. Estimating the β requires some approach, and statistical learning theory shows us that different approaches have different biases. Since our metric for evaluating performance on the regressions is OLS, $||Y - X\beta||^2$ is the natural loss function to minimize to recover the β. If the system is overdetermined, (n>d), there is not guaranteed to exist a β that satisfies $Y = X\beta$. The normal equations yield the β that minimizes squared loss-- $\beta = (X^T X)^{-1} X^T Y$. But if $d >> n$, then taking the inverse of $X^T X$ becomes less computationally feasible. If the system is underdetermined, n<d, the solution is possibly not unique.

One approach in calculating R^2 for the purposes of classifying the datasets is to use 1) $\beta = (X^T X)^{-1} X^T$ if n>d, and 2) $\beta = X^T (X X^T)^{-1}$ if n<d. The tradeoff is that while 1) minimizes squared loss, 2) minimizes the squared sum of the components of β s.t. $Y = X\beta$. So the predictor would be arbitrarily biased depending on whether n>d or n<d.

Another approach to calculate R^2 for the purposes of classifying the datasets is to use Gurobi to solve ridge regression, because we know that this scales for large datasets. We can then interpret the OCT results with an understanding that ridge regression biases towards solutions that ... We use the latter for the feature included in OCT. We provide a range for ρ and select the best via cross validation.

In [ ]:

In [136]:
```
using DataFrames, Plots, StatsBase, JuMP, Gurobi, DataFrames, CSV, Linea
rAlgebra, Random, Statistics
```

In [137]:
```
data = CSV.read("Final_Metadata_Reduced.csv", categorical=true);
```

In [138]:
```
# Split data
function splitter(data)
    data_shuffled = data[shuffle(1:end), :]

    # Split X and Y data
    Y_shuff = categorical(data_shuffled[:, 15])
    X_shuff = data_shuffled[:, 3:14]

    # Get indices for splitting your data
    n = size(data_shuffled)[1]
    train_ind = Int(round(.7*n))
    test_ind = n

    # Split X data based on indices
    X_train = X_shuff[1:train_ind, :]
    X_test = X_shuff[train_ind+1:n, :]

    # Split Y data based on indices
    Y_train = Y_shuff[1:train_ind]
    Y_test = Y_shuff[train_ind+1:n]

    return(X_train, X_test, Y_train, Y_test)
end
```

Out[138]:
```
splitter (generic function with 1 method)
```

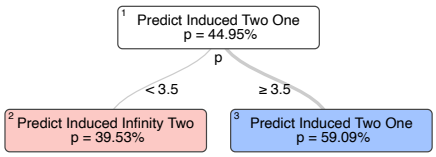
In [139]:
```
X_train, X_test, Y_train, Y_test = splitter(data);
```

Create initial tree

In [140]:
```
# Create initial tree
tree = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp=0.1, minb
ucket=10)

# Fit tree
IAI.fit!(tree, X_train, Y_train)
```

Out[140]:

Collapse Expand Save PNG Save HTML



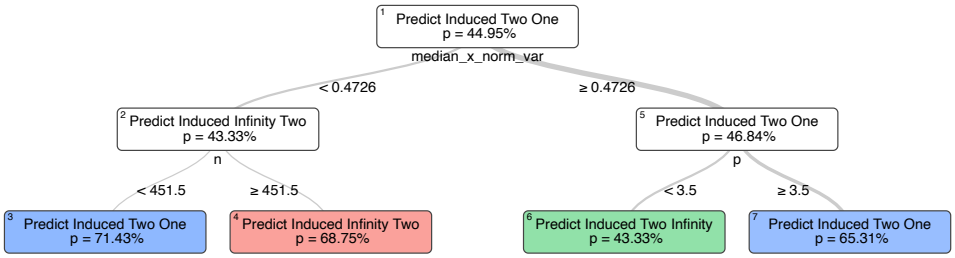
In [ ]:
```
# Create initial tree
tree = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp=0.1, minb
ucket=10)

# Fit tree
IAI.fit!(tree, select(X_train, [:p]), Y_train)
```

OCT with validation

In [143]:
```
best_tree = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp=0.00
1, minbucket=10)

grid = IAI.GridSearch(best_tree,
    max_depth=1:5,
    minbucket=[10,20],
    cp=[0.01, 0.1]
)

IAI.fit!(grid, X_train, Y_train)

best_mod = IAI.get_learner(grid)
```

Out[143]:

Collapse Expand Save PNG Save HTML



In [144]:
```
# Fit tree and measure performance
IAI.fit!(best_mod, X_train, Y_train)

min_score_test = IAI.score(best_mod, X_test, Y_test, criterion=:misclass
ification)
max_score_test = min_score_test

min_score_train = IAI.score(best_mod, X_train, Y_train, criterion=:miscl
assification)
max_score_train = min_score_train

max_score_test
```

Out[144]:
```
0.46808510638297873
```

Loop through a bunch of models to find accuracy statistics

In [117]:
```
accuracy_vec_train = []
accuracy_vec_test = []
```

Out[117]:
```
0-element Array{Any,1}
```

In [118]:
```
# Loop through to find score range
for i in 1:100
    X_train, X_test, Y_train, Y_test = splitter(data)

    tmp_tree = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp=
0.001, minbucket=10)

    tmp_grid = IAI.GridSearch(tmp_tree,
        max_depth=1:5,
        minbucket=[10,20],
        cp=[0.01, 0.1]
    )

    IAI.fit!(tmp_grid, X_train, Y_train)
    best_tmp = IAI.get_learner(tmp_grid)

    tmp_score_test = IAI.score(best_tmp, X_test, Y_test, criterion=:misc
lassification)
    tmp_score_train = IAI.score(best_tmp, X_train, Y_train, criterion=:m
isclassification)

    append!(accuracy_vec_train, tmp_score_train)
    append!(accuracy_vec_test, tmp_score_test)

    # Max and min test score accuracy
    if tmp_score_test < min_score_test
        min_score_test = tmp_score_test
    end
    if tmp_score_test > max_score_test
        max_score_test = tmp_score_test
    end

    # Max and min train score accuracy
    if tmp_score_train < min_score_train
        min_score_train = tmp_score_train
    end
    if tmp_score_train > max_score_train
        max_score_train = tmp_score_train
    end
end
```

In [130]:
```
print("Min train accuracy: ", min_score_train, "\n",
        "Max train accuracy: ", max_score_train, "\n",
        "Mean train accuracy: ", mean(accuracy_vec_train), "\n",
        "Train accuracy variance: ", var(accuracy_vec_train), "\n",
        "Min test accuracy: ", min_score_test, "\n",
        "Max test accuracy: ", max_score_test, "\n",
        "Mean test accuracy: ", mean(accuracy_vec_test), "\n",
        "Test accuracy variance: ", var(accuracy_vec_test)
        )
```

```
Min train accuracy: 0.45871559633027525
Max train accuracy: 0.7339449541284404
Mean train accuracy: 0.5810091743119267
Train accuracy variance: 0.0046318840283994705
Min test accuracy: 0.17021276595744683
Max test accuracy: 0.574468085106383
Mean test accuracy: 0.407872340425532
Test accuracy variance: 0.006072997974310789
```

OCT with all 7 uncertainty sets

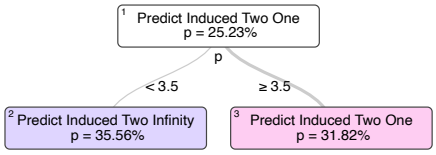
In [122]:
```
data_full = CSV.read("Final_Metadata3.csv", categorical=true);
```

In [123]:
```
X_train_full, X_test_full, Y_train_full, Y_test_full = splitter(data_ful
l);
```

In [145]:
```
best_tree_full = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp
=0.001, minbucket=10)

grid_full = IAI.GridSearch(best_tree_full,
    max_depth=1:5,
    minbucket=[10,20],
    cp=[0.01, 0.1]
)

IAI.fit!(grid_full, X_train_full, Y_train_full)

best_mod_full = IAI.get_learner(grid_full)
```

Out[145]:

Collapse Expand Save PNG Save HTML



In [146]:
```
# Fit tree and measure performance
IAI.fit!(best_mod_full, X_train_full, Y_train_full)

min_score_test_full = IAI.score(best_mod_full, X_test_full, Y_test_full,
criterion=:misclassification)
max_score_test_full = min_score_full_test

min_score_train_full = IAI.score(best_mod_full, X_train_full, Y_train_fu
ll, criterion=:misclassification)
max_score_train_full = min_score_full_train
```

Out[146]:
```
0.34042553191489366
```

In [127]:
```
acc_vec_train_full = []
acc_vec_test_full = []
```

Out[127]:
```
0-element Array{Any,1}
```

In [128]:
```
# Loop through to find score range
for i in 1:100
    X_train_full, X_test_full, Y_train_full, Y_test_full = splitter(data
_full)

    tmp_tree = IAI.OptimalTreeClassifier(random_seed=1,max_depth=5, cp=
0.001, minbucket=10)

    tmp_grid = IAI.GridSearch(tmp_tree,
        max_depth=1:5,
        minbucket=[10,20],
        cp=[0.01, 0.1]
    )

    IAI.fit!(tmp_grid, X_train_full, Y_train_full)
    best_tmp = IAI.get_learner(tmp_grid)

    tmp_score_test = IAI.score(best_tmp, X_test_full, Y_test_full, crite
rion=:misclassification)
    tmp_score_train = IAI.score(best_tmp, X_train_full, Y_train_full, cr
iterion=:misclassification)

    append!(acc_vec_train_full, tmp_score_train)
    append!(acc_vec_test_full, tmp_score_test)

    # Max and min test score accuracy
    if tmp_score_test < min_score_test_full
        min_score_test_full = tmp_score_test
    end
    if tmp_score_test > max_score_test_full
        max_score_test_full = tmp_score_test
    end

    # Max and min train score accuracy
    if tmp_score_train < min_score_train_full
        min_score_train_full = tmp_score_train
    end
    if tmp_score_train > max_score_train_full
        max_score_train_full = tmp_score_train
    end
end
```

In [131]:
```
print("Min train accuracy: ", min_score_train_full, "\n",
        "Max train accuracy: ", max_score_train_full, "\n",
        "Mean train accuracy: ", mean(acc_vec_train_full), "\n",
        "Train accuracy variance: ", var(acc_vec_train_full), "\n",
        "Min test accuracy: ", min_score_test_full, "\n",
        "Max test accuracy: ", max_score_test_full, "\n",
        "Mean test accuracy: ", mean(acc_vec_test_full), "\n",
        "Test accuracy variance: ", var(acc_vec_test_full)
        )
```

```
Min train accuracy: 0.23423423423423428
Max train accuracy: 0.5765765765765766
Mean train accuracy: 0.41513513513513517
Train accuracy variance: 0.005913661409156904
Min test accuracy: 0.12765957446808507
Max test accuracy: 0.4042553191489362
Mean test accuracy: 0.23829787234042563
Test accuracy variance: 0.003603257564325923
```

15.095 Project - Multinomial Logistic Regression

Desiree Waugh, Charlie Chimento

12/9/2019

```
# Libraries
library(glmnet)
library(caret)
library(lubridate)
library(caTools)
library(gbm)
library(dplyr)
library(nnet)
```

```
# Read in data
data <- read_csv("Final_Metadata_Reduced.csv")
data$uncertainty_set <- as.factor(data$uncertainty_set)
```

```
# Perform testing procedure 100 times
accuracy_vec_train <- c()
accuracy_vec_test <- c()

# Initial values
train_percent <- 0.7
data_shuffled <- data[sample(nrow(data)),]
data_train <- data_shuffled[1:(round(train_percent*nrow(data_shuffled))),]
data_test <- data_shuffled[(round(train_percent*nrow(data_shuffled))+1):nrow(data_shuffled),]

# Model with important variables
log_mod <- multinom(uncertainty_set ~ n+p+mean_cor, data=data_train)
predictions_test <- predict(log_mod, newdata=data_test)
predictions_train <- predict(log_mod, newdata=data_train)

confusion_matrix_test <- table(predictions_test, data_test$uncertainty_set)
min_accuracy_test <- (confusion_matrix_test[1,1] + confusion_matrix_test[2,2] +
                        confusion_matrix_test[3,3]) / nrow(data_test)
max_accuracy_test <- min_accuracy_test

confusion_matrix_train <- table(predictions_train, data_train$uncertainty_set)
min_accuracy_train <- (confusion_matrix_train[1,1] + confusion_matrix_train[2,2] +
                         confusion_matrix_train[3,3]) / nrow(data_train)
max_accuracy_train <- min_accuracy_train

for (i in 1:100){
  # Split into training and testing sets
  train_percent <- 0.7
  data_shuffled <- data[sample(nrow(data)),]
  data_train <- data_shuffled[1:(round(train_percent*nrow(data_shuffled))),]
  data_test <- data_shuffled[(round(train_percent*nrow(data_shuffled))+1):nrow(data_shuffled),]

  # Model with important variables
  log_mod <- multinom(uncertainty_set ~ n+p+mean_cor, data=data_train)
  predictions_test <- predict(log_mod, newdata=data_test)
```

```
  predictions_train <- predict(log_mod, newdata=data_train)

  confusion_matrix_test <- table(predictions_test, data_test$uncertainty_set)
  tmp_accuracy_test <- (confusion_matrix_test[1,1] + confusion_matrix_test[2,2] +
                          confusion_matrix_test[3,3]) / nrow(data_test)
  accuracy_vec_test <- c(accuracy_vec_test, tmp_accuracy_test)

  confusion_matrix_train <- table(predictions_train, data_train$uncertainty_set)
  tmp_accuracy_train <- (confusion_matrix_train[1,1] + confusion_matrix_train[2,2] +
                           confusion_matrix_train[3,3]) / nrow(data_train)
  accuracy_vec_train <- c(accuracy_vec_train, tmp_accuracy_train)

  # Max and min test accuracy
  if (tmp_accuracy_test < min_accuracy_test){
    min_accuracy_test <- tmp_accuracy_test
  }
  if (tmp_accuracy_test > max_accuracy_test){
    max_accuracy_test <- tmp_accuracy_test
  }

  # Max and min train accuracy
  if (tmp_accuracy_train < min_accuracy_train){
    min_accuracy_train <- tmp_accuracy_train
  }
  if (tmp_accuracy_train > max_accuracy_train){
    max_accuracy_train <- tmp_accuracy_train
  }
}

cat("Min train accuracy: ", min_accuracy_train)
cat("Max train accuracy: ", max_accuracy_train)
cat("Mean train accuracy: ", mean(accuracy_vec_train))
cat("Train accuracy variance: ", var(accuracy_vec_train))
cat("Min test accuracy: ", min_accuracy_test)
cat("Max test accuracy: ", max_accuracy_test)
cat("Mean test accuracy: ", mean(accuracy_vec_test))
cat("Test accuracy variance: ", var(accuracy_vec_test))
```

```
### Baseline Models ###
# Baseline - Reduced Dataset
baseline_table <- data %>%
  group_by(uncertainty_set) %>%
  summarise(Percent = n() / nrow(data))

# Baseline - Full Dataset
data_full <- read_csv("Final_Metadata3.csv")
data_full$uncertainty_set <- as.factor(data_full$uncertainty_set)

baseline_full <- data_full %>%
  group_by(uncertainty_set) %>%
  summarize(Percent = n() / nrow(data_full))
```