PREDICTING FUNGAL PROTEIN SUBCELLULAR LOCATION

by

James D. Munyon

Submitted in Partial Fulfillment of the Requirements
for the Degree of

BACHELOR OF SCIENCE

in the
Department of Mathematics and Statistics

YOUNGSTOWN STATE UNIVERSITY
May, 2015

ABSTRACT

Proteins perform many functions within the cells of organisms, and these functions are closely related to their subcellular location: where in a cell they reside. Protein sequences are entering databases faster than their subcellular locations can be empirically measured, so there is a need for predictors that can accurately predict protein subcellular locations. One numerical representation of the amino acid composition of proteins is called pseudo amino acid composition, turning a long string of amino acids which make up a protein into a length 20 (or greater) vector (whose first 20 values are the normalized occurrence frequencies of the 20 "standard" amino acids). Through this transformation, using a benchmark dataset of 3002 fungal proteins, initial decision tree methods of random forests, Adaboost, SAMME, and bagging will be applied to protein data to establish "baseline" predictive performance results, and then some prediction methods found in the literature, support vector machines and the covariant discriminant algorithm, will be applied as well. We will find that support vector machines improve over all of the decision tree methods, with the covariant discriminant algorithm giving an even further improvement, with potential room to perform better in and of itself in the near future. Future plans in terms of alternative data transformation techniques will be discussed as well.

ACKNOWLEDGEMENTS

I would like to thank Dr. Andy Chang (Department of Mathematics and Statistics, Youngstown State University) and Dr. Jack Min (Department of Biological Sciences, Youngstown State University) for their suggestion of this research and their help in forwarding literature material and data, Kofi Atta Neizer-Ashun for helping to make the hurdle over a significant early obstacle in data transformation, Dr. Joseph Palardy (Department of Economics, Youngstown State University) for continued suggestions of better practices in using R, all who work on and develop R and R packages (this work would be nowhere without your commitment to the availability of open source software), and of course my friends and family for always putting up with me.

Contents

1 Introduction

Kuo-Chen Chou and Hong-Bin Shen's 2007 review paper, "Recent progress in protein subcellular location prediction", provides an excellent introduction to this area of research, and both of the authors are associated with many papers in the literature. The review article provides a good basis for much of the introduction section here.

First of all, we must mention biological cells. "...the cell is deemed to be the most basic structural and functional unit of all living organisms and often is called a 'building block of life' " [1]. Many different components, probably known as organelles, perform specialized tasks inside cells. It is mostly these organelles that we will come to term as "subcellular locations". We can briefly describe the functions of some organelles, specifically those which

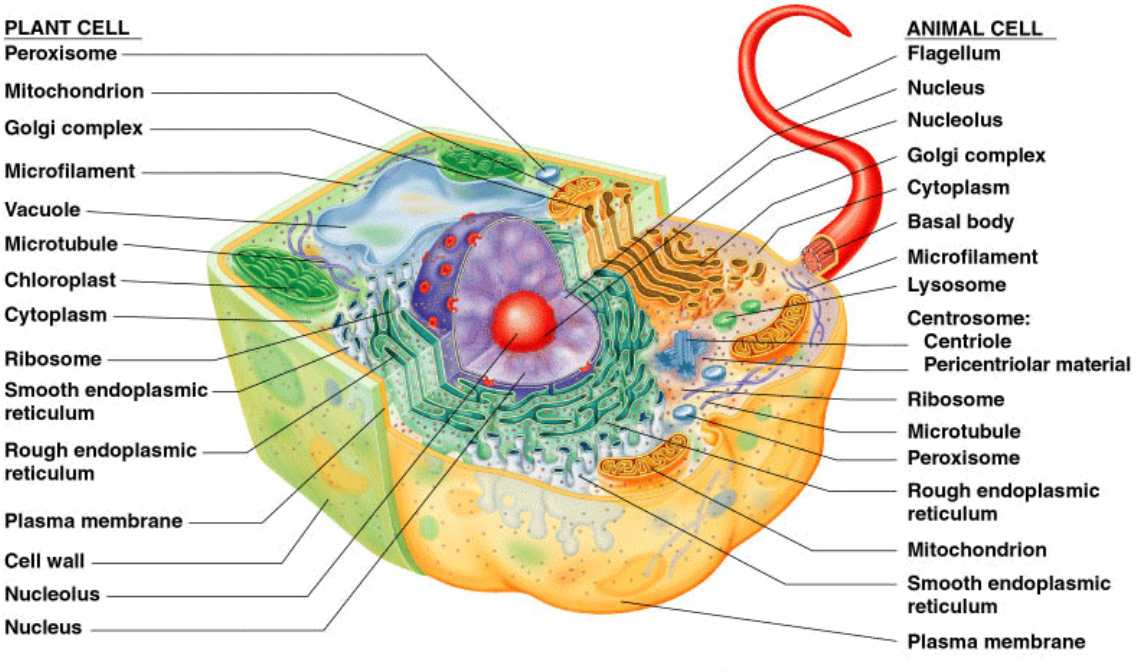

Figure 1: Cell organelles

will appear later in the paper as potential locations for proteins to be predicted to. The cytoplasm is the "jelly" that holds the cell together, and other organelles are suspended in it. The cytoskeleton is a network of filaments that branches through the cytoplasm, serving many functions. The endoplasmic reticulum synthesizes proteins and lipids and transports

proteins. The Golgi apparatus modifies and stores products of the endoplasmic reticulum. The mitochondria is associated with cellular respiration. The nucleus contains chromosomes and is basically the "powerhouse" or brain of the cell. The peroxisome turns peroxide into water. The plasma membrane surrounds the cell as a lipid layer that controls what goes in and out of the cell. And the vacuole is a sac that holds materials such as water [2].

It is the case that many of the functions that sustain cell life are due to the work of proteins. There may be around one billion proteins in an average cell, and since these proteins are so important to cell function, it is a must to understand their functions and residencies. While physical biochemical experiments can be conducted to measure the subcellular locations of proteins, these are expensive, take time, and are impractical in the modern age of fast entry of protein data into databases. Thus, many proteins can be found in databases that either lack subcellular location annotation, or that have annotation with "uncertain labels", i.e. the annotation was not through experimental observation. If progress cannot be made, the gap between newly found proteins and their subcellular locations will grow. Quoting the paper, "To use these newly found proteins for basic research and drug discovery in a timely manner, it is highly desired to develop an effective method to bridge such a gap. During the past 15 years, a variety of predictors have been developed to deal with the challenge" [1]. The paper then goes on to discuss predictors that the authors consider distinguishable from others based on some special features.

Now that we understand the general reasons why many proteins lack subcellular location annotation and why such annotation is desired, let us shift back to understanding proteins some more. The building blocks of proteins are amino acids, and there are 20 that are found in proteins, from alanine through valine. The biological activities of a protein are mainly determined by the amino acids that constitute it, and these activities are most cell processes as well as the catalyzation of reactions in cells [3]. For the purposes of this research, the amino acid sequences of proteins will be our raw data, subject to transformation via Chou's pseudo amino acid composition (which will be described in a further section), which then will result in our cleaned-up data which will be subject to actual methods/algorithms and analysis. There also exist other ways of representing proteins as data, which can be used alone, or in conjunction with something like pseudo amino acid composition. These

include full sequential representation, a protein's functional domain, GO database space [1], position-specific scoring matrices [4], and others.

In terms of our actual predictors (models), we will have models that are built with protein training data (including annotation for known subcellular locations). For then analyzing the predictive accuracy of the models, testing data (which omits known location annotation) will be fed through them, and what will result is a list of predicted locations for each protein in the testing data. These predictions are then compared with the corresponding known locations, and accuracy can be measured in a variety of ways, including by location. If a predictor has a high accuracy on the testing data, there is hope that this predictive ability will translate to other proteins out in the protein population, and that in the future, accurate predictors can actually be used by researchers to augment experimental observation of protein locations, and give reasonable predictions for proteins that truly lack subcellular location annotation.

2 Literature Review

There exist many papers in the literature on the topic of predicting protein subcellular location/localization. Since three different aspects of problem-solving exist (choosing how to represent the protein as data, choosing how to transform this raw data, and choosing prediction algorithms/methods), there exist numerous combinations of approaches for solving the problem of protein subcellular location prediction. For example, Chou and Shen's review paper focuses on the following transformation and prediction combinations: amino acid composition with the covariant discriminant algorithm (CD); pseudo amino acid composition with CD, K nearest neighbors (KNN), and optimized evidence theoretic K nearest neighbors (OET-KNN); FunD with KNN and OET-KNN; and GO with KNN and OET-KNN, where amino acid composition is a "simpler" form of pseudo amino acid composition which will be seen in a further section, FunD and GO involve proteins being coded with zeros and ones depending on "hits" against databases, the covariant discriminant algorithm is a method used in this paper, and the KNN/OET-KNN methods involve, respectively, a protein being assigned to the location that its K nearest neighbors are a part of, and a more complicated variant thereof [1]. Chou and Shen were also willing to consider the more difficult multiclass problem: a problem where proteins should be predicted to more than one location, of which many papers in the literature do not cover.

In other papers, we find many other different approaches. Some methods end up as computational tools that can be found on websites/webservers, or that can be downloaded as executables. Some of these include SignalP, WoLF PSORT, Phobius, TargetP, TMHMM, FragAnchor, and PS-Scan [5]. In this author's opinion, however, many of these tools contain a steep learning curve inherent in their usage, as their internal methods/algorithmic steps and output can be quite cryptic and difficult for non-experts to read and interpret. Also for example, TargetP only predicts proteins to one of three possible locations, and this could be considered "not enough" or "too broad" in terms of possibilities [6].

Otherwise, we can also find, as examples: the covariant discriminant algorithm predicting for apoptosis proteins [7]; pseudo amino acid composition in conjunction with the Lyapunov index, Bessel function, Chebyshev filter and complexity measure factor [8,9];

position-specific scoring matrices, principal component analysis, and support vector machines [4]; log-odds sequence logos [10] and a host of other method combinations. The coverage of just these mentioned papers shows the many different possible approaches to predicting locations of these proteins, and certainly there exist plenty more methods that still need to be attempted or created.

3 Methodology

The data for this project was obtained from the UniProt Knowledgebase (UniProtKB), in the "reviewed entries" subset, named "Swiss-Prot". Proteins from fungal organisms were the focus of this project. To define a good "benchmark" dataset, several subsetting criteria which are found in many papers in the literature were followed:

- Proteins must be reviewed and have annotation for subcellular location
- Proteins must only have annotation for one location (as the multi-class prediction problem is more difficult, and not the subject of this project)
- Protein location evidence must be experimental (not inferential)
- Proteins must not be fragments (amino acid sequences must start with methionine)
- Proteins must not have unknown amino acids anywhere in their sequences
- Proteins must have 100 or more amino acids in their sequences (this qualifies as "sufficiently long")

After subsetting by the previous rules, "50/50" BLASTClust was then applied to the proteins (if two or more proteins are at least 50% similar over at least 50% of their lengths, they are considered as part of a cluster, and only one of them is randomly chosen to be kept). After all of this subsetting, 3002 fungal proteins remained, their annotated subcellular locations being the following: cytoplasm for 770 proteins, cytoskeleton for 95 proteins, endoplasmic reticulum (ER) for 38 proteins, endoplasmic reticulum membrane (ER membrane) for 247 proteins, Golgi apparatus for 24 proteins, Golgi apparatus membrane for 75 proteins, mitochondria for 365 proteins, mitochondria membrane for 187 proteins, nucleus for 952 proteins, nuclear membrane for 24 proteins, peroxisome for 8 proteins, peroxisome membrane for 11 proteins, plasma membrane for 10 proteins, secreted for 82 proteins, vacuole for 14 proteins, and vacuole membrane for 100 proteins, for a total of 16 different subcellular locations being represented. For initial methods, the data was divided into a 70% testing set and a 30% training set; however, later methods would take advantage of cross-validation (either ten-fold or jackknife) and thus use the entire dataset for training

and testing. Then the data was appropriately transformed via Chou's pseudo amino acid composition, using a correlation factor of $\lambda = 15$. This left us with a data matrix of 3002 rows and 35 columns (3002 observations of 35 variables), where 35 comes from 20 standard amino acids + 15 interaction terms. Chou's pseudo amino acid composition is as follows [1]: define a protein $\mathbf{P}$ with L amino acid residues by

$$\mathbf{P} = R_1 R_2 ... R_L, \tag{1}$$

so that the protein is represented by a string of L letters, the ith letter corresponding to the ith amino acid in it's sequence. Then it's pseudo amino acid composition is formulated as

$$\mathbf{P} = [p_1, p_2, ..., p_{20}, p_{20+1}, ..., p_{20+\lambda}]^T, \ (\lambda < L), \tag{2}$$

with an individual component as

$$p_u = \begin{cases} \dfrac{f_u}{\sum_{i=1}^{20} f_i + w \sum_{k=1}^{\lambda} \tau_k} & , 1 \leq u \leq 20 \\ \dfrac{w\tau_{u-20}}{\sum_{i=1}^{20} f_i + w \sum_{k=1}^{\lambda} \tau_k} & , 20+1 \leq u \leq 20+\lambda, \end{cases} \tag{3}$$

which means that (referring to [1] for more details), $\mathbf{P}$ is represented as its 20 occurrence frequencies of the standard amino acids, plus $\lambda = 15$ "interaction terms" which use biological equations to take some of the sequence order information of the protein into account, with all $20 + 15 = 35$ of these values finally normalized to sum to one.

3.1 Decision Trees

Decision trees are structures that show paths of possible decisions and the outcomes from following these paths. These structures can be built as predictive models for the purposes of predicting class labels of new testing data after being built on appropriate training data, and many algorithms exists that build decision tree models in different ways. See [11] for a (slightly old but) good overview of decision tree learning, and [12] for a description of the "CART" (Classification and Regression Trees) algorithm, which creates a binary decision tree by repeatedly splitting a node into two child nodes, starting with a root node that

contains all of the training data [12]. These ideas will make sense after looking at an example decision tree. Figure 2 is based on the question "Should we play tennis today?",

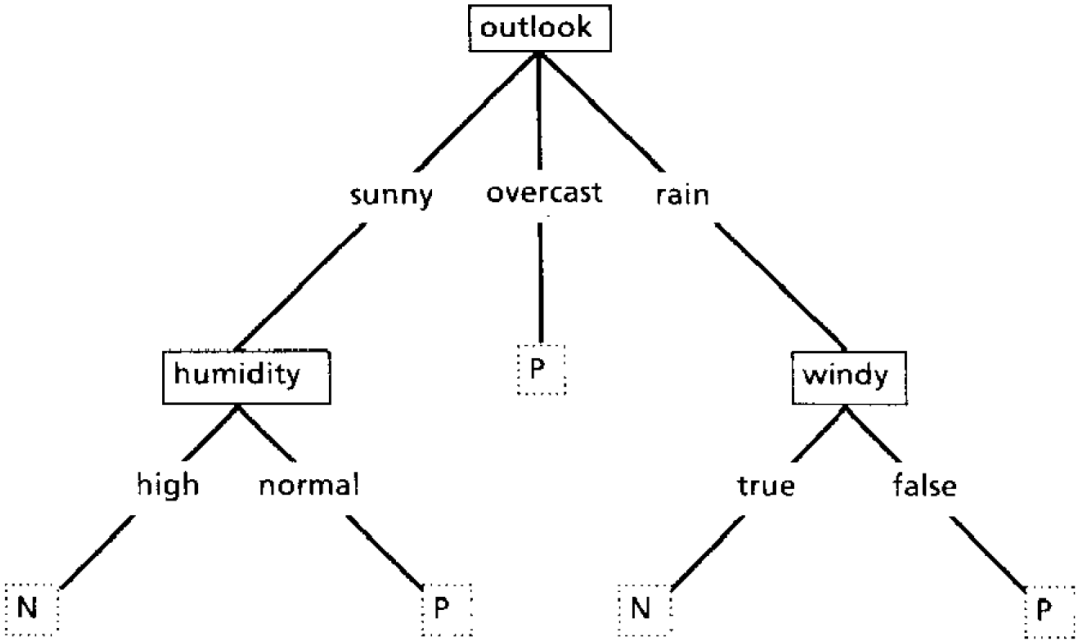

Figure 2: A decision tree

the answer depending on the weather. The answers, from left to right, are

- "If it's sunny with high humidity, then no."
- "If it's sunny with normal humidity, then yes."
- "If it's overcast, then yes."
- "If it's rainy and windy, then no."
- "If it's rainy but not windy, then yes."

Now imagine the tree to not be a personal decision-making tool, but instead something else. Imagine if many tennis players are surveyed, asked whether or not they would play tennis or not due to the weather, where the variables being recorded are "General Weather" (sunny, overcast, or rain), "Humidity" (high or normal), and "Windy" (true or false). The decision tree then classifies specific combinations of weather parameters as being "good for tennis" or as being "bad for tennis". In reality, not all weather combinations would be

perfectly predicted by the tree (some people may say they would play tennis on sunny, humid days even though most wouldn't), but hopefully most would be. The decision tree could then be a good challenger against other statistical classification methods. The next four described methods involve ensembles of decision trees: many different trees, each one generally learned in some slightly different way than the others, such that the trees in the ensemble are then averaged over to obtain final classification predictions for testing data.

3.2 Random Forests

From the abstract of Leo Breiman's famous 2001 paper "Random Forests" [13]: "Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest". For a slightly-modified version of Breiman's definition: "A random forest is a classifier consisting of a collection of tree-structured classifiers [using] independent identically distributed random vectors [for determining the different variables involved in the creation of each individual tree] and each tree casts a unit vote for the most popular class [for each data point]...". In simple terms, a random forest is a combination of many individual tree predictors, where each decision tree is built using a randomly-selected subset of our variables and a bootstrapped sample of our training data. Each tree then outputs a class prediction for each data point, and the mode of the predictions for a data point becomes its final prediction. Breiman's paper goes into detail and covers many topics such as theoretical justification, bounds on error rates, comparisons to adaptive boosting, implementation, etc. A quick suggestion for the success of random forests is that an upper bound for the error rate of a decision tree ensemble can be shown to depend on the correlations between the trees in the ensemble, and that a random forest can have low between-tree correlations.

3.3 Adaptive Boosting

As mentioned in [14], boosting can be a successful technique for solving a two-class classification problem (in the case of a multi-class problem, the problem is split up into several two-class problems). A well known specific algorithm is "AdaBoost" as given by Freund and Schapire in 1997. In AdaBoost, the idea is to combine many weak classifiers in an

appropriate linear combination such that the resulting final classifier is very accurate. Start with all training data points equally-weighted and build a classifier (such as a decision tree). For data points that are misclassified, increase their weights, and for points that have been correctly classified, decrease their weights. Then build a new classifier. This classifier will hopefully classify the higher-weighted data points better than the previous classifier. Repeat this process many, many times, then combine all of the classifiers in a linear combination to obtain final classifications for all training data points. Let training data be defined as $(\mathbf{x}_1, c_1), ..., (\mathbf{x}_n, c_n)$ where a vector of predictor variables $\mathbf{x}_i \in \mathbb{R}^n$ and a response variable $c_i \in 1, 2, ..., K$ (is qualitative, taking on one of a finite number of values). The AdaBoost algorithm is then as follows:

- Initialize observation weights as $w_i = 1/n, i = 1, 2, ..., n$
- For $m = 1$ to M (with M being the number of classifiers to build):
  - Fit a classifier $T^{(m)}(x)$ to the training data using weights w_i
  - Compute $err^{(m)} = \sum_{i=1}^{n} w_i \times \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right) / \sum_{i=1}^{n} w_i$
  - Compute $\alpha^{(m)} = log\left(\frac{1 - err^{(m)}}{err^{(m)}}\right)*$
  - Set $w_i \leftarrow w_i \times exp\left(\alpha^{(m)} \times \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right)\right), i = 1, 2, ..., n$
  - Re-normalize w_i
- Output $C(x) = \underset{x}{\text{argmax}} \sum_{m=1}^{M} \alpha^{(m)} \times \mathbb{I}(T^{(m)}(x) = k)$

AdaBoost is considered to be very accurate for two-class problems, however, it's potential lack of applicability to multi-class problems has led to the next method to be seen.

3.4 SAMME

SAMME (Stagewise Additive Modeling using a Multi-class Exponential Loss Function) is the same as AdaBoost, with the only difference being condition * from before. In SAMME, $\alpha^{(m)} = log\left(\frac{1 - err^{(m)}}{err^{(m)}}\right) + log(K - 1)$. Note that when $K = 2$, SAMME reduces to AdaBoost. Otherwise, the one change can make SAMME a better alternative to AdaBoost in multi-class problems, with details given in [the Zhu et al. paper].

3.5 Bagging

Random forests weren't Breiman's first foray into ensembles of decision trees. In 1996, his paper "Bagging Predictors" [15] introduced the idea of combining multiple predictors such that each one used a bootstrapped sample of the training data, to improve performance and the potential instability inherent in some procedures. The word "bagging" is a stand-in for "bootstrap aggregating", and the terms are used interchangeably. One of the sections in his paper deals with the application of bagging to decision trees. We can explain our implementation with simplicity. We build multiple decision trees, where each tree uses a bootstrapped sample of our training data. Thus, for an individual classifier, a data point from the training data may appear in our bootstrapped training data once, more than once, or not at all. After training our many, many trees, the final prediction for a data point is the mode of the predictions given for that data point by all of the trees.

3.6 Support Vector Machines

Some papers in the literature, including very recent papers, suggest the use of support vector machines (SVMs) as appropriate models for classification [4,16]. Note that the method only considers two-class classification problems "by default", but modifications can be made for a problem like ours that involves 16 classes (locations). By [17], the algorithm can be defined. Consider testing data points (of one of two possible classes) as labeled pairs of the form $(\mathbf{x}_i, y_i), i = 1, ..., l$ with $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$. Each $\mathbf{x}_i$ is the vector of variable values associated with a specific data point, and its associated y_i indicates which of the two classes the data point is a member of. Taken together, each $(\mathbf{x}_i, y_i)$ is one row of our testing data matrix. The following quadratic optimization problem is then solved:

$$\begin{aligned} &\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\xi_i \\ &\text{subject to } y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ &\qquad\qquad\qquad\qquad \xi_i \geq 0. \end{aligned} \tag{4}$$

What we have are the training vectors $\mathbf{x}_i$ being mapped to a higher, possibly infinite dimensional space by the function ϕ. The idea is that, in a higher dimensional space, data points of different classes may be linearly separable using a hyperplane, where they weren't linearly separable originally. Or if they still aren't linearly separable, they are at least much more linearly separable than they originally were. When training points are (closely or fully) linearly separable, testing data points can then be predicted based on which side of the separating hyperplane they fall on. Back in (4), $C > 0$ and the ξ_i indicate that perfect separation may not be attained as previously mentioned. Also important is $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ (the kernel function), where we use the radial basis function (RBF)

$$K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma ||\mathbf{x}_i - \mathbf{x}_j||^2), \gamma > 0. \tag{5}$$

C and γ are the two parameters for the researcher to choose (or, practically, find the best pair using cross-validation). Note that our problem involves 16 possible classes, not just two, so to use the SVM method, $(16)(16-1)/2 = 120$ "one vs. one" models must be created, where each one considers just training data points of two classes and finds the best separation between just those two. A voting scheme then determines the overall final prediction for a data point.

3.7 Covariant Discriminant Algorithm

This (somewhat older) method is mentioned in [1,18], and can be described as follows by [1]. Consider $(\mathbf{P}_1, \mathbf{P}_2, ..., \mathbf{P}_N)$, a group of N proteins, from the M possible subcellular locations $(S_1, S_2, ..., S_M)$. In our problem, $N = 3002$ and $M = 16$. Now consider any location subset S_m. It's uth protein is represented as

$$P_m^u = [p_{m,1}^u p_{m,2}^u ... p_{m,20}^u ... p_{m,20+\lambda}^u]^T, \tag{6}$$

each entry being one of the $20+\lambda$ values from the protein's pseudo amino acid composition. Also define the standard vector of S_m as

$$\bar{\mathbf{P}}_m = [\bar{p}_{m,1} \bar{p}_{m,2} ... \bar{p}_{m,20} ... \bar{p}_{m,20+\lambda}]^T, \tag{7}$$

each entry being an average pseudo amino acid composition value for all of the proteins in S_m. Consider $\bar{\mathbf{P}}_m$ to be the "standard" or "average" protein in subcellular location m. Now let $\mathbf{P}$ be a query protein whose location we wish to predict. We want to consider the similarity between $\mathbf{P}$ and each $\bar{\mathbf{P}}_m$. Define our similarity measure as follows:

$$F(\mathbf{P}, \bar{\mathbf{P}}_m) = D^2_{Mah}(\mathbf{P}, \bar{\mathbf{P}}_m) + ln|\mathbf{C}_m|, \tag{8}$$

where

$$D^2_{Mah}(\mathbf{P}, \bar{\mathbf{P}}_m) = (\mathbf{P} - \bar{\mathbf{P}}_m)^T \mathbf{C}_m^{-1} (\mathbf{P} - \bar{\mathbf{P}}_m) \tag{9}$$

is the squared Mahalanobis distance between $\mathbf{P}$ and $\bar{\mathbf{P}}_m$. $\mathbf{C}_m$ is the $(20 + \lambda) \times (20 + \lambda)$ covariance matrix for S_m with entry

$$c^m_{i,j} = \frac{1}{N_m - 1} \sum_{u=1}^{N_m} \left(p^u_{m,i} - \bar{p}_{m,i}\right) \left(p^u_{m,j} - \bar{p}_{m,j}\right), \tag{10}$$

$\mathbf{C}_m^{-1}$ is its inverse, $|\mathbf{C}_m|$ is its determinant, and N_m is just the number of proteins in S_m. It should be noted that due to properties of the Pseudo Amino Acid Composition, any covariance matrix $\mathbf{C}_m$ will be singular and hence not have an inverse. The work-around will be the following "dimension-reducing" procedure: drop one of the $20 + \lambda$ variables (normalized amino acid frequencies or interaction terms) so that only $20 + \lambda - 1$ are considered. Then the covariance matrices become theoretically invertible (although it may still be difficult in practice), and progress can continue. It can also be shown that it doesn't matter which variable is dropped: $F(\mathbf{P}, \bar{\mathbf{P}}_m)$ values will end up the same no matter which variable is dropped [1]. In our case, $M = 16$ $F(\mathbf{P}, \bar{\mathbf{P}}_m)$ values will be computed for query protein $\mathbf{P}$, and the location associated with the smallest one will become the location prediction for $\mathbf{P}$.

4 Analysis

In the spirit of [19], all of our decision tree methods were run using either the data split into a 70% training set and a 30% testing set, or using ten-fold cross-validation on the entire dataset. In the training set of 2101 proteins, locations were as follows: cytoplasm for 526 proteins, cytoskeleton for 66 proteins, endoplasmic reticulum (ER) for 29 proteins, endoplasmic reticulum membrane (ER membrane) for 174 proteins, Golgi apparatus for 18 proteins, Golgi apparatus membrane for 50 proteins, mitochondria for 268 proteins, mitochondria membrane for 132 proteins, nucleus for 663 proteins, nuclear membrane for 17 proteins, peroxisome for 6 proteins, peroxisome membrane for 7 proteins, plasma membrane for 5 proteins, secreted for 58 proteins, vacuole for 9 proteins, and vacuole membrane for 73 proteins, and in the testing set of 901 proteins, locations were as follows: cytoplasm for 244 proteins, cytoskeleton for 29 proteins, endoplasmic reticulum (ER) for 9 proteins, endoplasmic reticulum membrane (ER membrane) for 73 proteins, Golgi apparatus for 6 proteins, Golgi apparatus membrane for 25 proteins, mitochondria for 97 proteins, mitochondria membrane for 55 proteins, nucleus for 289 proteins, nuclear membrane for 7 proteins, peroxisome for 2 proteins, peroxisome membrane for 4 proteins, plasma membrane for 5 proteins, secreted for 24 proteins, vacuole for 5 proteins, and vacuole membrane for 27 proteins. All analysis was done using R statistical software, except for the attempted calculation of some matrix inverses, which was done in MATLAB (to no avail).

4.1 Defining Some Terms

We shall quickly define some terms which shall soon appear with frequency. When discussing a method in full, (classification) accuracy = the percentage of observations in the testing data (or in the full dataset if using some form of cross-validation) that are correctly classified. When discussing a specific location X compared against all others in a method:

- TP = number of true positives (predicted to be in X and actually in X)
- FP = number of false positives (predicted to be in X but actually not in X)
- FN = number of false negatives (not predicted to be in X but actually in X)

- TN = number of true negatives (not predicted to be in X and actually not in X)
- Sensitivity = TP / (TP + FN)
- Specificity = TN / (FP + TN)
- Balanced accuracy = (sensitivity + specificity) / 2
- Matthews correlation coefficient (MCC) =

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

  - If the denominator is 0, then just remove it from the formula as the numerator will come out to 0 which is correct for interpretation

Balanced accuracy is a measure that will ensure that we don't give too much weight to either sensitivity or specificity (as a well-predicted location will have both of them relatively high anyway), and MCC is another "mixture" measure of performance, considered to be a value that summarizes well a 2×2 confusion matrix, and such that values between classes are comparable even if the classes are of different sizes [19], which they very much are in our case.

4.2 Random Forests

R's "randomForest" package contains implementations of Leo Breiman's original random forest algorithm. We built a forest of size 500 trees on our 70% training data and then used the model to predict subcellular locations for the 30% testing data. Recall that a random selection of variables is used to build each tree.

Table 1: Overall Statistics

Accuracy	0.47
95% CI	(0.43, 0.5)
No Information Rate	0.32
$p-value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.28

Table 2: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.45	0.76	0.41	0.79	0.27	0.12	0.3	0.6	0.2	244
Cytoskeleton	0	1	NA	0.97	0.03	0	0	0.5	0	29
ER	0	1	NA	0.99	0.01	0	0	0.5	0	9
ER (membrane)	0.52	0.95	0.47	0.96	0.08	0.04	0.09	0.73	0.45	73
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	6
Golgi apparatus (membrane)	0.04	1	1	0.97	0.03	0	0	0.52	0.2	25
Mitochondria	0.32	0.97	0.53	0.92	0.11	0.03	0.07	0.64	0.36	97
Mitochondria (membrane)	0.04	1	1	0.94	0.06	0	0	0.52	0.18	55
Nucleus	0.78	0.61	0.48	0.86	0.32	0.25	0.52	0.69	0.36	289
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	7
Peroxisome	0	1	NA	1	0	0	0	0.5	0	2
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	4
Plasma membrane	0	1	NA	0.99	0.01	0	0	0.5	0	5
Secreted	0.62	0.99	0.65	0.99	0.03	0.02	0.03	0.81	0.63	24
Vacuole	0	1	NA	0.99	0.01	0	0	0.5	0	5
Vacuole (membrane)	0	1	0	0.97	0.03	0	0	0.5	-0.01	27

Key:

Sn: Sensitivity, Sp: Specificity, PPV: Positive Predicted Value, NPV: Negative Predicted Value, Pr: Prevalence, DR: Detection Rate, DP: Detection Prevalence, BA: Balanced Accuracy, MCC: Matthews Correlation Coefficient, NObs: Number of Proteins

Tables 1 and 2 respectively show overall statistics and statistics by class for the predictive ability of the random forest model on the training data. Here and elsewhere, covariance matrices will be omitted in the interest of avoiding difficult-to-read/-understand tables. It can be seen that 47% of proteins in the training data are correctly predicted to their subcellular locations. Unfortunately for many locations, by sensitivities of 0 and specificities of 1, it is the case that no proteins are predicted to those locations, leaving less than half of the locations to absorb all of the predictions. By this, MCC values are 0 for those locations, which indicates a performance on par with that of random guessing. Stand-out locations for decent performance are cytoplasm, ER (membrane), mitochondria, nucleus, and secreted, in terms of balanced accuracy and MCC. Our best location is secreted: 66% sensitivity (about two-thirds correctly predicted), 99% specificity (virtually no incorrect predictions of secreted), balanced accuracy of 81%, and MCC of .63. The model performs alright but is too general to predict for locations that (for the most part) have few representatives in the testing data and overall.

4.3 Adaptive Boosting

R's "adabag" package contains implementations of Freund and Schapire's AdaBoost, Zhu's SAMME, and Breiman's bagging. For AdaBoost to stay consistent with random forest, we build an ensemble of 500 trees using a bootstrapped sample of our training data for each tree. Recall that data point weights will not be uniform after the first tree is created.

Table 3: Overall Statistics

Accuracy	0.43
95% CI	(0.39, 0.46)
No Information Rate	0.32
$p-value[ACC > NIR]$	2.3×10^{-11}
Kappa	0.23

Table 4: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.41	0.71	0.35	0.77	0.27	0.11	0.32	0.56	0.12	244
Cytoskeleton	0	1	NA	0.97	0.03	0	0	0.5	0	29
ER	0	1	NA	0.99	0.01	0	0	0.5	0	9
ER (membrane)	0.51	0.94	0.43	0.96	0.08	0.04	0.1	0.72	0.41	73
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	6
Golgi apparatus (membrane)	0.04	1	1	0.97	0.03	0	0	0.52	0.2	25
Mitochondria	0.37	0.92	0.36	0.92	0.11	0.04	0.11	0.65	0.29	97
Mitochondria (membrane)	0	1	NA	0.94	0.06	0	0	0.5	0	55
Nucleus	0.7	0.66	0.49	0.82	0.32	0.22	0.46	0.68	0.33	289
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	7
Peroxisome	0	1	NA	1	0	0	0	0.5	0	2
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	4
Plasma membrane	0	1	NA	0.99	0.01	0	0	0.5	0	5
Secreted	0.33	0.99	0.57	0.98	0.03	0.01	0.02	0.66	0.43	24
Vacuole	0	1	NA	0.99	0.01	0	0	0.5	0	5
Vacuole (membrane)	0	1	NA	0.97	0.03	0	0	0.5	0	27

By tables 3 and 4, adaptive boosting doesn't perform quite as well as random forests. Looking at secreted proteins for example, our specificity is the same, but our sensitivity is half of what it was in random forests, which this brings down balanced accuracy and MCC. To make a quick point about MCC, look at the performance for nucleus proteins. We have the highest sensitivity (70%), indicating that the model is most likely to correctly predict nucleus proteins. The balanced accuracy is also the second highest (68%), yet the MCC

(.33) isn't as high as that of two other locations (ER membrane, .41 and secreted, .43). This shows how MCC takes class size into account, almost as if saying "your testing data has many nucleus proteins, so it's less impressive for many of those to be correctly predicted; we're more impressed with the decent performance of ER membrane and secreted proteins, with their low counts in the testing data".

4.4 Ten-fold cross-validation

For an alternative validation method, we use ten-fold cross-validation on the whole dataset instead of using the training and testing set. This allows each protein to be training data for 90% of the time, while still getting a prediction. Base, simple intuition might suggest that we should expect better results with the cross-validation.

Table 5: Overall Statistics

Accuracy	0.42
95% CI	(0.39, 0.44)
No Information Rate	0.32
$p-value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.22

Table 6: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.37	0.74	0.33	0.77	0.26	0.1	0.29	0.56	0.11	770
Cytoskeleton	0	1	NA	0.97	0.03	0	0	0.5	0	95
ER	0	1	NA	0.99	0.01	0	0	0.5	0	38
ER (membrane)	0.53	0.92	0.37	0.96	0.08	0.04	0.12	0.72	0.38	247
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	24
Golgi apparatus (membrane)	0	1	NA	0.98	0.02	0	0	0.5	0	75
Mitochondria	0.24	0.95	0.39	0.9	0.12	0.03	0.07	0.59	0.23	365
Mitochondria (membrane)	0	1	NA	0.94	0.06	0	0	0.5	0	187
Nucleus	0.75	0.61	0.47	0.84	0.32	0.24	0.5	0.68	0.33	952
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	24
Peroxisome	0	1	NA	1	0	0	0	0.5	0	8
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	11
Plasma membrane	0	1	NA	1	0	0	0	0.5	0	10
Secreted	0.41	0.99	0.59	0.98	0.03	0.01	0.02	0.7	0.48	82
Vacuole	0	1	NA	1	0	0	0	0.5	0	14
Vacuole (membrane)	0	1	NA	0.97	0.03	0	0	0.5	0	100

Despite our expectation of the ten-fold cross-validation improving adaptive boosting, performance is very similar across many metrics.

4.5 SAMME

Recall that SAMME is supposed to be a better alternative to (modified to handle more than two-class cases) AdaBoost. Implementation is the same: 70/30 training/testing split, 500 trees built on bootstrapped samples of the training data.

Table 7: Overall Statistics

Accuracy	0.42
95% CI	(0.38, 0.45)
No Information Rate	0.32
$p - value[ACC > NIR]$	1.7×10^{-9}
Kappa	0.22

Table 8: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.57	0.62	0.35	0.79	0.27	0.15	0.43	0.59	0.16	244
Cytoskeleton	0	1	0	0.97	0.03	0	0	0.5	-0.01	29
ER	0	1	NA	0.99	0.01	0	0	0.5	0	9
ER (membrane)	0.45	0.95	0.46	0.95	0.08	0.04	0.08	0.7	0.41	73
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	6
Golgi apparatus (membrane)	0.04	1	0.33	0.97	0.03	0	0	0.52	0.11	25
Mitochondria	0.36	0.93	0.38	0.92	0.11	0.04	0.1	0.64	0.3	97
Mitochondria (membrane)	0.04	0.99	0.18	0.94	0.06	0	0.01	0.51	0.06	55
Nucleus	0.53	0.74	0.49	0.77	0.32	0.17	0.35	0.63	0.26	289
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	7
Peroxisome	0	1	NA	1	0	0	0	0.5	0	2
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	4
Plasma membrane	0	1	NA	0.99	0.01	0	0	0.5	0	5
Secreted	0.38	1	0.82	0.98	0.03	0.01	0.01	0.69	0.55	24
Vacuole	0	1	NA	0.99	0.01	0	0	0.5	0	5
Vacuole (membrane)	0.07	0.99	0.25	0.97	0.03	0	0.01	0.53	0.12	27

SAMME gives us some new best performers, but still overall we're failing to predict at all for many of the same locations.

By this point, we have implemented all of the methods found in [19], so far finding random forests being slightly better than AdaBoost and SAMME. However, it felt necessary

to check bagging as well, as another common decision tree ensemble method.

4.6 Bagging

Implementation is the same as the previous methods. Bagging can be thought of as similar to AdaBoost, except that individual trees are independent.

Table 9: Overall Statistics

Accuracy	0.41
95% CI	(0.37, 0.45)
No Information Rate	0.32
$p - value[ACC > NIR]$	6×10^{-9}
Kappa	0.21

Table 10: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.41	0.69	0.33	0.76	0.27	0.11	0.34	0.55	0.1	244
Cytoskeleton	0	1	NA	0.97	0.03	0	0	0.5	0	29
ER	0	1	NA	0.99	0.01	0	0	0.5	0	9
ER (membrane)	0.48	0.94	0.41	0.95	0.08	0.04	0.09	0.71	0.39	73
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	6
Golgi apparatus (membrane)	0	1	0	0.97	0.03	0	0	0.5	-0.01	25
Mitochondria	0.37	0.92	0.36	0.92	0.11	0.04	0.11	0.65	0.29	97
Mitochondria (membrane)	0	1	NA	0.94	0.06	0	0	0.5	0	55
Nucleus	0.66	0.66	0.48	0.81	0.32	0.21	0.44	0.66	0.3	289
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	7
Peroxisome	0	1	NA	1	0	0	0	0.5	0	2
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	4
Plasma membrane	0	1	NA	0.99	0.01	0	0	0.5	0	5
Secreted	0.33	0.99	0.57	0.98	0.03	0.01	0.02	0.66	0.43	24
Vacuole	0	1	NA	0.99	0.01	0	0	0.5	0	5
Vacuole (membrane)	0	1	0	0.97	0.03	0	0	0.5	-0.01	27

Bagging ends up being arguably our worst method: lowest overall accuracy of 41%, only five locations get predictions at all (mostly those with many observations, as no surprise), and two locations (Golgi apparatus membrane and vacuole membrane) have negative MCC values, indicating slightly worse that just random prediction for those locations (in fact, it also means that their specificities are actually slightly less than 100%, the 100% being due to rounding). It seems that we can't just repeatedly build similar models (just using slightly

different training sets each time due to bootstrapping) to aggregate with any mentionable accuracy.

4.7 Ten-fold cross-validation

And for the ten-fold cross-validation...

Table 11: Overall Statistics

Accuracy	0.42
95% CI	(0.39, 0.44)
No Information Rate	0.32
$p - value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.22

Table 12: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.39	0.73	0.33	0.78	0.26	0.1	0.3	0.56	0.11	770
Cytoskeleton	0	1	NA	0.97	0.03	0	0	0.5	0	95
ER	0	1	NA	0.99	0.01	0	0	0.5	0	38
ER (membrane)	0.53	0.92	0.36	0.96	0.08	0.04	0.12	0.72	0.37	247
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	24
Golgi apparatus (membrane)	0	1	NA	0.98	0.02	0	0	0.5	0	75
Mitochondria	0.21	0.95	0.38	0.9	0.12	0.03	0.07	0.58	0.21	365
Mitochondria (membrane)	0	1	NA	0.94	0.06	0	0	0.5	0	187
Nucleus	0.74	0.62	0.48	0.84	0.32	0.24	0.49	0.68	0.34	952
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	24
Peroxisome	0	1	NA	1	0	0	0	0.5	0	8
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	11
Plasma membrane	0	1	NA	1	0	0	0	0.5	0	10
Secreted	0.43	0.99	0.62	0.98	0.03	0.01	0.02	0.71	0.51	82
Vacuole	0	1	NA	1	0	0	0	0.5	0	14
Vacuole (membrane)	0	1	NA	0.97	0.03	0	0	0.5	0	100

Essentially the same conclusion as before: bagging is not optimal for our purposes.

4.8 Summarizing Decision Trees

By this point it was realized that our decision tree methods had pretty much hit a limit in terms of improvement, and that there was very little difference between the 70/30 training/testing split vs. the ten-fold cross-validation. Random forests slightly edges out other

ensemble methods, with a classification rate of 47% against an average of about 42% otherwise, but this difference arguably isn't too major. All "statistics by class" tables exhibit similarity, mostly showing that cytoplasm, ER (membrane), mitochondria, nucleus, and secreted proteins get essentially all of the predictions, with only nuclear proteins consistently getting accurate predictions at a rate greater than 50%. Also take note that it is generally the case that the locations listed have relatively more representatives in the data than the other locations. This isn't surprising, as a model is penalized less having a maxim such as "when in doubt, predict nucleus", but this means that proteins with few representatives will have much more difficulty in being accurately predicted.

It was now time to explore other, more "literature-specific" methods, including the use of jackknife validation.

4.9 Support Vector Machines

For this method, jackknife validation is used to assess model performance. In jackknife validation, build a predictive model using all but one of your data points, and then use this model to predict the class for the one left-out data point. Repeat this process for each data point being used as the left-out data point. In our case, 3002 models are created as our benchmark dataset has 3002 proteins. Jackknife validation allows every protein to have its location predicted where the greatest possible amount of information builds the model that is doing the predicting.

As previously mentioned, for an SVM model using the RBF kernel, we have two parameters that we must choose, C and λ. [17] recommends a combined cross-validation and "grid-search" strategy to find the optimal parameters for a given problem. In the strategy, designate various (C, λ) pairs to test, and choose the pair that gives the best cross-validation accuracy using an SVM model that uses that pair. The paper suggests using exponentially-growing sequences of values for C and λ: try $C \in \{2^{-5}, 2^{-3}, ..., 2^{15}\}$ and $\lambda \in \{2^{-15}, 2^{-13}, ..., 2^{3}\}$. We choose ten-fold cross-validation for determining the accuracy of each (C, λ)-paired SVM model. Once the best (C, λ) pair is determined, can then use jackknife validation to acquire a prediction for each protein in the benchmark dataset, and measure the final performance of the method.

Table 13: Overall Statistics

Accuracy	0.5
95% CI	(0.48, 0.53)
No Information Rate	0.32
$p - value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.35

Table 14: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.53	0.74	0.41	0.82	0.26	0.14	0.33	0.64	0.25	770
Cytoskeleton	0.01	1	0.33	0.97	0.03	0	0	0.5	0.05	95
ER	0	1	NA	0.99	0.01	0	0	0.5	0	38
ER (membrane)	0.52	0.96	0.56	0.96	0.08	0.04	0.08	0.74	0.5	247
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	24
Golgi apparatus (membrane)	0.05	1	0.44	0.98	0.02	0	0	0.53	0.15	75
Mitochondria	0.45	0.95	0.56	0.93	0.12	0.05	0.1	0.7	0.44	365
Mitochondria (membrane)	0.24	0.98	0.48	0.95	0.06	0.01	0.03	0.61	0.31	187
Nucleus	0.72	0.72	0.54	0.85	0.32	0.23	0.42	0.72	0.41	952
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	24
Peroxisome	0	1	NA	1	0	0	0	0.5	0	8
Peroxisome (membrane)	0.36	1	1	1	0	0	0	0.68	0.6	11
Plasma membrane	0	1	NA	1	0	0	0	0.5	0	10
Secreted	0.8	0.99	0.75	0.99	0.03	0.02	0.03	0.9	0.77	82
Vacuole	0	1	NA	1	0	0	0	0.5	0	14
Vacuole (membrane)	0.17	0.99	0.47	0.97	0.03	0.01	0.01	0.58	0.27	100

We can already see a noticeable increase in improvement over the decision tree methods, in the overall classification rate of 50% (not much larger than for random forests, but so compared against the others), and in table 14 we see more locations actually receiving predictions than before. Nucleus continues to perform well, and the largest surprise is for secreted proteins: 80% specificity and 99% sensitivity(!) (meaning that 80% of secreted proteins are predicted correctly, and 99% of non-secreted proteins are not predicted as secreted) is leaps and bounds above any class performance we've seen so far, giving a balanced accuracy of 90%. Clearly secreted proteins are clustered together, away from other proteins in the projected-into 35 or higher dimensional space. This may not be a surprise in a biological sense since secreted means "these proteins are secreted outside of the cell", which is clearly physically different than being in any location inside the cell, but

this is the first time a model has picked this up from the data. We also have some pretty good MCC values for locations such as secreted, peroxisome membrane, and ER membrane.

4.10 Covariant Discriminant Algorithm

For this method we also use jackknife validation, as it lends itself to the method very nicely. For each protein, "simply" compare it to the average protein of each location using the previously-mentioned similarity measure (8), which involves a non-Euclidean measure of distance. The location associated with the smallest similarity measure value for a protein becomes that protein's location prediction.

As previously mentioned, we had the dimension-reducing work-around to be able to have invertible covariance matrices; recall their entries from (10). However, in practice, we still could not find meaningful inverses for some of our covariance matrices, and they corresponded to the following subcellular locations: Golgi apparatus, nuclear membrane, peroxisome, peroxisome membrane, plasma membrane, and vacuole. 91 proteins in our benchmark dataset correspond to one of these locations. In practice, those six locations have less than 35 representatives in our dataset, thus their covariance matrices are not invertible "by default", and we can't determine similarity measures for any protein and an average protein from any of those locations, so we set those similarity measures as arbitrarily large to ensure that those locations will not be predicted. As a consequence, we know that the 91 mentioned proteins will certainly be predicted incorrectly. Nonetheless, we can still attempt the method and see how well it performs, and the invertibility problem can be tackled in the future (more data?).

Table 15: Overall Statistics

Accuracy	0.61
95% CI	(0.59, 0.63)
No Information Rate	0.32
$p-value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.53

Table 16: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.47	0.9	0.61	0.83	0.26	0.12	0.2	0.68	0.4	770
Cytoskeleton	0.92	0.94	0.34	1	0.03	0.03	0.09	0.93	0.54	95
ER	1	1	1	1	0.01	0.01	0.01	1	1	38
ER (membrane)	0.7	0.96	0.62	0.97	0.08	0.06	0.09	0.83	0.63	247
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	24
Golgi apparatus (membrane)	0.92	0.98	0.51	1	0.02	0.02	0.04	0.95	0.68	75
Mitochondria	0.65	0.94	0.61	0.95	0.12	0.08	0.13	0.8	0.58	365
Mitochondria (membrane)	0.78	0.94	0.47	0.98	0.06	0.05	0.1	0.86	0.57	187
Nucleus	0.57	0.91	0.74	0.82	0.32	0.18	0.24	0.74	0.52	952
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	24
Peroxisome	0	1	NA	1	0	0	0	0.5	0	8
Peroxisome (membrane)	0	1	NA	1	0	0	0	0.5	0	11
Plasma membrane	0	1	NA	1	0	0	0	0.5	0	10
Secreted	0.99	1	0.93	1	0.03	0.03	0.03	0.99	0.96	82
Vacuole	0	1	NA	1	0	0	0	0.5	0	14
Vacuole (membrane)	0.93	0.97	0.49	1	0.03	0.03	0.06	0.95	0.66	100

First of all, notice our new best classification accuracy of 61%, mildly surprising since we knew that proteins from six different locations would certainly be predicted incorrectly. But for an even larger surprise, see table 16. In near opposition to table 14 (for the SVMs), instead of proteins with many representatives being particularly classified well, we find proteins with few representatives being classified well, some at very high rates such as cytoskeleton, Golgi apparatus membrane, secreted, and vacuole proteins. Notice, for secreted proteins, a 99% sensitivity, a 100% specificity, and an MCC value of .96, indicating that all but one of the 82 secreted proteins were predicted correctly, and that no other proteins were falsely predicted as being secreted. Even better off are ER proteins, the first time we can say that proteins in a location are predicted perfectly: sensitivity, specificity, balanced accuracy, and MCC are all 100% (1). This performance is extraordinary, and one can only wonder how proteins in the six unpredictable locations would have fared, as we notice that all of those locations have few representatives. Hopefully in the future those matrix inverses can be calculated once more representatives can be obtained for those locations.

4.11 Mixture Strategy

One can't help but wonder (since one method predicts well for "common" locations and the other predicts well for "uncommon" locations) if the following post-hoc "mixture" strategy for combining support vector machines and the covariant discriminant algorithm might be appropriate: for each location $i \in \{1, ..., 16\}$,

$$\text{method\_for\_predicting}_i = \begin{cases} CDA & \text{, if } NObs_i \leq k \text{ and } i \notin \{5, 10, 11, 12, 13, 15\} \\ SVM & \text{, otherwise} \end{cases} \tag{11}$$

for some appropriate integer k? For "small" or "large" k we'll be very close to pure CDA or pure SVM respectively, but for some "middle of the road" k we expect the mixture to perform better. We went ahead and checked this using $k = NObs_i$, $i = 1, ..., 16$. Note the second condition for the CDA method being used: this ensures that the location isn't Golgi apparatus, nuclear membrane, peroxisome, peroxisome membrane, plasma membrane, or vacuole (if it is, we know that the CDA method will be wrong for sure, so we use the SVM method regardless of the size of $NObs_i$). The best performer is $k = 365$, which corresponds to the following: using the SVM method for cytoplasm and nucleus proteins, and also (by second condition) Golgi apparatus, nuclear membrane, peroxisome, peroxisome membrane, plasma membrane, and vacuole proteins, and using the CDA method for all other proteins. So now we can see how well our post-hoc mixture strategy performs.

Table 17: Overall Statistics

Accuracy	0.67
95% CI	(0.66, 0.69)
No Information Rate	0.32
$p - value[ACC > NIR]$	2.2×10^{-16}
Kappa	0.59

Table 18: Statistics by Class

	Sn	Sp	PPV	NPV	Pr	DR	DP	BA	MCC	NObs
Cytoplasm	0.53	0.86	0.57	0.84	0.26	0.14	0.24	0.69	0.4	770
Cytoskeleton	0.92	0.99	0.78	1	0.03	0.03	0.04	0.95	0.84	95
ER	1	1	1	1	0.01	0.01	0.01	1	1	38
ER (membrane)	0.7	0.98	0.77	0.97	0.08	0.06	0.08	0.84	0.71	247
Golgi apparatus	0	1	NA	0.99	0.01	0	0	0.5	0	24
Golgi apparatus (membrane)	0.92	0.99	0.73	1	0.02	0.02	0.03	0.96	0.81	75
Mitochondria	0.65	0.97	0.72	0.95	0.12	0.08	0.11	0.81	0.64	365
Mitochondria (membrane)	0.78	0.98	0.7	0.98	0.06	0.05	0.07	0.88	0.72	187
Nucleus	0.72	0.82	0.65	0.86	0.32	0.23	0.35	0.77	0.52	952
Nucleus (membrane)	0	1	NA	0.99	0.01	0	0	0.5	0	24
Peroxisome	0	1	NA	1	0	0	0	0.5	0	8
Peroxisome (membrane)	0.36	1	1	1	0	0	0	0.68	0.6	11
Plasma membrane	0	1	NA	1	0	0	0	0.5	0	10
Secreted	0.99	1	0.87	1	0.03	0.03	0.03	0.99	0.93	82
Vacuole	0	1	NA	1	0	0	0	0.5	0	14
Vacuole (membrane)	0.93	0.99	0.78	1	0.03	0.03	0.04	0.96	0.85	100

Our Golgi apparatus, nuclear membrane, peroxisome, plasma membrane, and vacuole proteins are never predicted, a consequence of the fact that they were regulated to SVM prediction which doesn't perform to well on those locations with few proteins. But otherwise, performance is very pleasing in our post-hoc mixture method. We have an overall accuracy of 67%, five different locations with balanced accuracies >90%, and corresponding high MCC values.

5 Conclusions

Sufficiently many methods have been attempted to facilitate discussion relating to the predictive strength of methods on proteins of different locations. For our decision tree methods, predictive performance is simply not competitive with the later methods, in terms of overall classification accuracy, and ability on a wide variety of locations. The methods only even "guaranteed" predictions for five locations (cytoplasm, ER membrane, mitochondria, nucleus, and secreted) at all, mainly in the top locations in terms of number of observations found in either the testing data, or entire dataset in the cross-validation cases. In any particular case, the method was really only notably accurate on nucleus proteins as well, which helped to raise the classification accuracy, but didn't allow the model to generalize to other locations.

Our later methods (support vector machines and the covariant discriminant algorithm) fared better, particularly the latter. SVMs saw double as many locations receiving predictions than in the worst of the decision tree cases, saw the first 50% accuracy hit, and saw locations other than nucleus starting to perform well, such as cytoplasm, ER membrane, mitochondria, and secreted. And with CDA, improvement was even larger: 61% classification accuracy, locations with many proteins still performing at least decently, and, for the first time, many locations with few proteins being predicted almost perfectly, like cytoskeleton, Golgi apparatus membrane, secreted, and vacuole membrane. The only drawback was the five locations that were "out of the game" due to the covariance matrix invertibility problem. The proposed post-hoc mixture method hoped to fix this issue by defaulting proteins from those locations to being predicted by SVMs, and to also split predictions based on the number of proteins in the dataset for each location. It may only be something that can be considered "after the fact", yet accuracy raises to 67%, all non-zero MCC values are respectable, and we have several amazing performers (cytoskeleton, Golgi apparatus membrane, secreted, and vacuole membrane proteins). Our only disappointment is that Golgi apparatus, nuclear membrane, peroxisome, plasma membrane, and vacuole proteins continue to receive no predictions at all. They only represent $\frac{80}{3002} = 2.66\%$ of the dataset, but since they all have few counts, we would have the expectation of CDA to perform well

on them if only the aforementioned issue could be remedied. Like many problems that come out when working with data data, we could always do better with some more data.

All in all we are pleased with our results, as we saw a general trend towards improvement with our later methods, and expect continued improvement in the future.

6 Future Work

Future work shall consist of, among other endeavors, more investigation into the covariance matrices of the covariant discriminant algorithm (and periodic snooping of the protein database to see if we can find more proteins in those six mentioned locations), as well as exploration of alternative data transformation techniques (other than Chou's pseudo amino acid composition), such as using position-specific scoring matrices [4]. Through said matrices, proteins can be represented by a vector with entries being probabilities of certain amino acids biologically evolving into others. Then principal component analysis can be used to remove excess/unnecessary variables (which also helps with computation time), and a current good performer such as support vector machines or the covariant discriminant algorithm can then be applied.

7 References

[1] Chou, K.C. & Shen, H.B. (2007). Recent progress in protein subcellular location prediction. Analytical Biochemistry, 370, 1−16.

[2] List of Organelles. https://bioh.wikispaces.com/List+of+Organelles.

[3] (2003). The Chemistry of Amino Acids. http://www.biology.arizona.edu/biochemistry/problem_sets/aa/aa.html.

[4] Yao, Y.H., Shi, Z.X. & Dai, Q. (2014). Apoptosis Protein Subcellular Location Prediction Based on Position−Specific Scoring Matrix. Journal of Computational and Theoretical Nanoscience, 11, 2073−2078.

[5] Meinken, J., Asch, D.K., Neizer−Ashun, K.A., Chang, G.H., Cooper Jr, C.R. & Min, X.J. (2014). FunSecKB2: a fungal protein subcellular location knowledgebase. Computational Molecular Biology, 4(7), 1−17.

[6] (2005). Output format. http://www.cbs.dtu.dk/services/TargetP−1.1/output.php.

[7] Zhou, G.P. & Doctor, K. (2003). Subcellular Location Prediction of Apoptosis Proteins. Proteins: Structure, Function, and Genetics, 50, 44−48.

[8] Gao, Y., Shao, S., Xiao, X., Ding, Y., Huang, Y., Huang, Z. & Chou, K.C. (2005). Using pseudo amino acid composition to predict protein subcellular location: Approached with Lyapunov index, Bessel function, and Chebyshev filter. Amino Acids, 28, 373−376.

[9] Xiao, X., Shao, S., Ding, Y., Haung, Z., Haung, Y. & Chou, K. C. (2005). Using complexity measure factor to predict protein subcellular location. Amino Acids, 28, 57−61.

[10] Yu, Y.K., Capra, J.A., Stojmirovic, A., Landaman, D. & Altschul, S.F. (2015). Log–odds sequence logos. Bioinformatics, 31(3), 324–331.

[11] Quinlan, J.R. (1986). Induction of Decision Trees. Machine Learning, 1, 81–106.

[12] CART Algorithm, retreived from ftp://ftp.boulder.ibm.com/software/analytics/spss/support/Stats/Docs/Statistics/Algorithms/14.0/TREE-CART.pdf, 9:54 PM, 3/24/2015.

[13] Breiman, L. (2001). Random Forests. Machine Learning, 45(1), 5–32.

[14] Zhu, J., Rosset, S., Zou, H. & Hastie. T. (2006). Multi–class AdaBoost. Unpublished manuscript.

[15] Breiman, L. (1996). Bagging Predictors. Machine Learning, 24, 123–140.

[16] Hua, S. & Sun, Z. (2001). Support vector machine approach for protein subcellular localization prediction. Bioinformatics, 17(8), 721–728.

[17] Hsu, C.W., Chang, C.C. & Lin, C.J. (2010). A Practical Guide to Support Vector Classification. Unpublished manuscript.

[18] Chou, K.C. & Elrod, D.W. (1999). Protein subcellular location prediction. Protein Engineering, 12, 107–108.

[19] Neizer–Ashun, K.A. (2013). Prediction of Plant Protein Subcellular Locations. Unpublished manuscript.

[20] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

[21] Charif, D. & Lobry, J.R. (2007). seqinr: a contributed package to the R project for statistical computing devoted to biological sequences retrieval and analysis.

[22] Xiao, N., Xu, Q.S. & Cao, D.S. (2014). protr: Generating Various Numerical Representation Schemes of Protein Sequence. R package version 0.5-1.

[23] Hong, L. BioSeqClass: Classification for Biological Sequences. R package version 1.24.0.

[24] Liaw, A & Wiener, M. (2002). Classification and Regression by randomForest. R News, 2(3), 18-22.

[25] Bates, D., Maechler, M., Bolker, B. & Walker, S. (2014). lme4: Linear mixed-effects models using Eigen and S4. R package version 1.1-7, <URL: http://CRAN.R-project.org/package=lme4>.

[26] Kuhn, M. (2015). caret: Classification and Regression Training. R package version 6.0-41. http://CRAN.R-project.org/package=caret.

[27] Alfaro, E., Gamez, M. & Garcia, N. (2013). adabag: An R Package for Classification with Boosting and Bagging. Journal of Statistical Software, 54(2), 1-35. URL http://www.jstatsoft.org/v54/i02/.

[28] Ripley, B. (2014). tree: Classification and regression trees. R package version 1.0-35. http://CRAN.R-project.org/package=tree.

[29] Therneau, T., Atkinson, B. & Ripley, B. (2015). rpart: Recursive Partitioning and Regression Trees. R package version 4.1-9. http://CRAN.R-project.org/package=rpart.

[30] Torgo, L. (2010). Data Mining with R, learning with case studies Chapman and Hall/CRC. URL: http://www.dcc.fc.up.pt/~ltorgo/DataMiningWithR.

[31] Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A. & Leisch, F. (2014). e1071: Misc Functions of the Department of

Statistics (e1071), TU Wien. R package version 1.6-4. http://CRAN.R-project.org/package=e1071.

[32] Tang, S., Li, T., Cong, P., Xiong, W., Wang, Z. & Sun, J. (2013). PlantLoc: an accurate web server for predicting plant protein subcellular localization by substantiality motif. Nucleic Acids Research, 41, 441-447.

[33] Min, X.J. (2010). Evalutation of COmputational Methods for Secreted Protein Prediction in Different Eukaryotes. Journal of Proteomics & Bioinformatics, 3(4), 143-147.

[34] Niu, B., Jin, Y.H., Feng, K.Y., Lu, W.C., Cai, Y.D. & Li, G. Z. (2008). Using AdaBoost for the prediction of subcellular location of prokaryotic and eukaryotic proteins. Molecular Diversity, 12, 41-45.

[35] Chou, K.C. & Shen, H.B. (2010). A New Method for Predicting the Subcellular Localization of Eukaryotic Proteins with Both Single and Miltiple Sites: Euk-mPLoc 2.0. PLos ONE 5(4): e9931. doi:10.1371/journal.pone.0009931

[36] Meinken, J. & Min, X.J. (2012). Computational Prediction of Protein Subcellular Locations in Eukaryotes: an Experience Report. Computational Molecular Biology, 2(1), 1-7.

[37] Neizer-Ashun, K.A., Yu, F., Meinken, J., Min, X. & Chang, G. H. Prediction of Plant Protein Subcellular Locations. Unpublished manuscript.

[38] Li, H. (2013). Using the BioSeqClass Package. Unpublished manuscript.

[39] Nathan, M. A Multi-site Subcellular Localizer for Fungal Proteins. Unpublished manuscript.

[40] Meyer, D. (2014). Support Vector Machines - The Interface to libsvm in package e1071. Unpublished manuscript.

[41] Min, X.J. (2010). Evalutation of Computational Methods for Secreted Protein Prediction in Different Eukaryotes. Journal of Proteomics & Bioinformatics, 3(4), 143−147.

[42] (2003). Dr. Margaret Oakley Dayhoff. http://www.biology.arizona.edu/biochemistry/problem_sets/aa/Dayhoff.html.

8 Principal Commented Code

DATA READ-IN, SUBSETTING, CLEANUP, AND INITIAL METHODS:

```
### Warning: the reader may find some varible names strange


### First - install packages with install.packages()
### Note - installing BioSeqClass may be more difficult than just using install.
    packages("BioSeqClass")
### See http://www.bioconductor.org/packages/release/bioc/html/BioSeqClass.html
    for potential help
### Load packages - most of the following are used at some point:
library(seqinr)
library(protr)
library(party)
library(BioSeqClass)
library(randomForest)
library(lme4)
library(caret)
library(adabag)
library(tree)
library(rpart)
library(DMwR)
### Read in protein data, already mostly subsetted properly using online
    database
### (readFASTA may not work; may have to use read.fasta (they're from different
    packages)):
#stuff = readFASTA("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/benchmark.fasta", seqonly = FALSE)
stuff = read.fasta("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/benchmark.fasta", seqtype = "AA", as.string = TRUE)
vector = as.vector(stuff)
### Remove any proteins containing an X (unknown amino acid) in their sequences:
tacos = NULL
for(i in 1:length(vector)){
  if(grepl("X", vector[i]) == TRUE){
    tacos = c(tacos, i)
```

```
  }
}
vector = vector[-tacos]
### Shorten protein names to just their length 6 identifiers:
names(vector) = sapply(names(vector), function(x){
  substring(x, 4, 9)
})
### Set a seed for reproducability (since random sampling is coming up):
set.seed(1349)
### Read in output text file from "50/50" BLASTClust, where the one or more
    proteins on each line represent a specific cluster:
scan = scan("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    blasted.txt", what = character(), sep = "\n")
### Create an R list, each list element a cluster:
list = list()
for(i in 1:length(scan)){
  list[[i]] = unlist(strsplit(scan[i], " "))
}
### Shorten protein names to just their length 6 identifiers:
for(i in 1:length(list)){
  for(j in 1:length(list[[i]])){
    list[[i]][j] = substring(list[[i]][j], 4, 9)
  }
}
### For each cluster in the list, randomly select just one protein from the
    cluster:
proteins = NULL
for(i in 1:length(list)){
  proteins[i] = list[[i]][sample(1:length(list[[i]]), 1)]
}
### Remove proteins that weren't previously selected:
blasted_outta_here = NULL
for(i in 1:length(vector)){
  if((names(vector)[i] %in% proteins) == FALSE){
    blasted_outta_here = c(blasted_outta_here, i)
  }
```

```
}
vector = vector[-blasted_outta_here]
### Read in list of known protein locations:
locations = read.csv("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/locations_unique.csv", header = FALSE, col.names = c("Protein
    ", "Location"))
### Transform protein sequences by means of Pseudo Amino Acid Composition:
PAA = featurePseudoAAComp(vector, d = 15, w = 0.05)
### Save results to machine:
write.csv(PAA, "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv")
### Read results back in:
PAA = read.csv("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv", header = TRUE, row.names = 1)
### Reorder data alphabetically by protein identifier:
PAA = PAA[order(row.names(PAA)),]
### Remove location listings for proteins that aren't now in our dataset:
subcellular = rep("word", nrow(PAA))
a = NULL
for (i in 1:nrow(locations)){
  if (locations[i,1] %in% rownames(PAA) == FALSE){
    a = c(a,i)
  }
}
locations = locations[-a,]
### Combine known locations and the PAA values into a data frame:
subcellular = as.factor(as.character(locations$Location))
PAA = data.frame(subcellular, PAA)
### Save data frame to machine:
write.csv(PAA, "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv")
### End of preprocessing
########################
### Beginning of analysis
### "Refresh" seed - not really necessary:
set.seed(0)
```

```
### Read data back in:
DATA = read.csv("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv", header = TRUE, row.names = 1)
### Set a seed for reproducability (since random sampling is coming up):
set.seed(69)
### Take a sample of 70% of the numbers in 1 through the number of proteins in
    the dataset:
nums = sample(1:nrow(DATA), floor(.7*nrow(DATA)))
### Create 70% training set (and ensure that at least one protein from each
    location is in it
### (need to do this so models can know that each location is a "possibility"):
train = DATA[nums,]
while (0 %in% summary(train$subcellular) == TRUE){
  nums = sample(1:nrow(DATA), floor(.7*nrow(DATA)))
  train = DATA[nums,]
}
### Create 30% testing set:
test = DATA[-nums,]
### Random Forests
### Create Random Forest model with training data - 500 trees is default:
RF = randomForest(formula = subcellular ~ ., data = train, importance = TRUE,
    proximity = TRUE)
### Variable importance plot:
varImp_RF = varImpPlot(RF)
### Obtain predictions for testing data:
predicted_RF = predict(RF, newdata = test)
### Save actual locations of testing data:
actual = test[,1]
### Calculate error rate of classification:
error_RF = 1 - sum(predicted_RF == actual)/nrow(test)
### Create confusion matrix and performance statistics for each location:
confusion_RF = confusionMatrix(data = predicted_RF, reference = actual)
### Adaptive Boosting
### Set some "appropriate" parameter options for the algorithm:
cntrl = rpart.control(maxdepth = 6, minsplit = 0, cp = -1)
### Create adaptive boosting model with training data, using 500 trees:
```

```
ada = boosting(formula = subcellular ~ ., data = train, mfinal = 500, coeflearn
    = "Freund", boos = TRUE, control = cntrl)
### Variable importance plot:
varImp_ada = barplot(ada$importance[order(ada$importance, decreasing = TRUE)],
    ylim = c(0,20), main = "Variables Relative Importance", col = "lightblue")
### Obtain predictions for testing data:
predicted_ada = predict(object = ada, newdata = test, newmfinal = 500)
### Calculate error rate of classification:
error_ada = 1 - sum(predicted_ada$class == actual)/nrow(test)
### Create confusion matrix and performance statistics for each location:
predicted_ada_class = as.factor(predicted_ada$class)
levels(predicted_ada_class) = c(levels(predicted_ada_class), levels(actual))
predicted_ada_class = factor(predicted_ada_class, levels(predicted_ada_class)[
    order(levels(predicted_ada_class))])
confusion_ada = confusionMatrix(data = predicted_ada_class, reference = actual)
### Adaptive Boosting with ten-fold cross-validation
### Create adaptive boosting with ten-fold cross-validation model with full
    dataset, using (50 trees in each "fold")*(10 "folds") = 500 total trees:
ada_cv = boosting.cv(formula = subcellular ~ ., data = DATA, v = 10, boos = TRUE
    , mfinal = 50, coeflearn = "Freund", control = rpart.control(maxdepth = 10))
### Variable importance plot (gives an error):
#varImp_ada_cv = barplot(ada_cv$importance[order(ada_cv$importance, decreasing =
     TRUE)], ylim = c(0,20), main = "Variables Relative Importance", col = "
    lightblue")
### Save actual locations of full dataset:
ACTUAL = DATA[,1]
### Calculate error rate of classification:
error_ada_cv = 1 - sum(ada_cv$class == ACTUAL)/nrow(DATA)
### Create confusion matrix and performance statistics for each location:
ada_cv_class = as.factor(ada_cv$class)
levels(ada_cv_class) = c(levels(ada_cv_class), levels(actual))
ada_cv_class = factor(ada_cv_class, levels(ada_cv_class)[order(levels(
    ada_cv_class))])
confusion_ada_cv = confusionMatrix(data = ada_cv_class, reference = ACTUAL)
### SAMME
### Create SAMME model with training data, using 500 trees
```

```
samme = boosting(formula = subcellular ~ ., data = train, mfinal = 500,
    coeflearn = "Zhu", boos = TRUE, control = cntrl)
### Variable importance plot:
varImp_samme = varImp_samme = barplot(samme$importance[order(samme$importance,
    decreasing = TRUE)], ylim = c(0,20), main = "Variables Relative Importance",
     col = "lightblue")
### Obtain predictions for testing data:
predicted_samme = predict(object = samme, newdata = test, newmfinal = 500)
### Calculate error rate of classification:
error_samme = 1 - sum(predicted_samme$class == actual)/nrow(test)
### Create confusion matrix and performance statistics for each location:
predicted_samme_class = as.factor(predicted_samme$class)
levels(predicted_samme_class) = c(levels(predicted_samme_class), levels(actual))
predicted_samme_class = factor(predicted_samme_class, levels(
    predicted_samme_class)[order(levels(predicted_samme_class))])
confusion_samme = confusionMatrix(data = predicted_samme_class, reference =
    actual)
### Save Random Forests results to machine:
write.csv(as.data.frame.matrix(confusion_RF$table), file = "C:/Users/jdmunyon/
    Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/RF_confusion.csv
    ")
write.csv(as.data.frame.matrix(confusion_RF$byClass), file = "C:/Users/jdmunyon/
    Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/RF_stats.csv")
write.table(error_RF, file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/results/50_50/RF_stats.csv", append = TRUE)
### Save Adaptive Boosting results to machine:
write.csv(as.data.frame.matrix(confusion_ada$table), file = "C:/Users/jdmunyon/
    Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/ada_confusion.csv
    ")
write.csv(as.data.frame.matrix(confusion_ada$byClass), file = "C:/Users/jdmunyon
    /Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/ada_stats.csv")
write.table(error_ada, file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/results/50_50/ada_stats.csv", append = TRUE)
### Save Adaptive Boosting with ten-fold cross-validation results to machine:
write.csv(as.data.frame.matrix(confusion_ada_cv$table), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
```

```
    ada_cv_confusion.csv")
write.csv(as.data.frame.matrix(confusion_ada_cv$byClass), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    ada_cv_stats.csv")
write.table(error_ada_cv, file = "C:/Users/jdmunyon/Desktop/Dropbox/
    Senior_Project/data_and_code/results/50_50/ada_cv_stats.csv", append = TRUE)
### Save SAMME results to machine:
write.csv(as.data.frame.matrix(confusion_samme$table), file = "C:/Users/jdmunyon
    /Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/samme_confusion.
    csv")
write.csv(as.data.frame.matrix(confusion_samme$byClass), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    samme_stats.csv")
write.table(error_samme, file = "C:/Users/jdmunyon/Desktop/Dropbox/
    Senior_Project/data_and_code/results/50_50/samme_stats.csv", append = TRUE)
### And now a few more methods
### Bagging (Bootstrap Aggregating)
### Create bagging model with training data, using 500 trees:
bagging = bagging(formula = subcellular ~ ., data = train, mfinal = 500, control
     = cntrl)
### Variable importance plot:
varImp_bagging = barplot(bagging$importance[order(bagging$importance, decreasing
     = TRUE)], ylim = c(0,20), main = "Variables Relative Importance", col = "
    lightblue")
### Obtain predictions for testing data:
predicted_bagging = predict(object = bagging, newdata = test, newmfinal = 500)
### Calculate error rate of classification:
error_bagging = 1 - sum(predicted_bagging$class == actual)/nrow(test)
### Create confusion matrix and performance statistics for each location:
predicted_bagging_class = as.factor(predicted_bagging$class)
levels(predicted_bagging_class) = c(levels(predicted_bagging_class), levels(
    actual))
predicted_bagging_class = factor(predicted_bagging_class, levels(
    predicted_bagging_class)[order(levels(predicted_bagging_class))])
confusion_bagging = confusionMatrix(data = predicted_bagging_class, reference =
    actual)
```

```
### Bagging with ten-fold cross-validation
### Create bagging with ten-fold cross-validation model using full dataset,
    using (50 trees in each "fold")*(10 "folds") = 500 total trees:
bagging_cv = bagging.cv(formula = subcellular ~ ., data = DATA, v = 10, mfinal =
     50, control = rpart.control(maxdepth = 10))
### Variable importance plot (gives an error):
#varImp_bagging_cv = barplot(bagging_cv$importance[order(bagging_cv$importance,
    decreasing = TRUE)], ylim = c(0,20), main = "Variables Relative Importance",
     col = "lightblue")
### Calculate error rate of classification:
error_bagging_cv = 1 - sum(bagging_cv$class == ACTUAL)/nrow(DATA)
### Create confusion matrix and performance statistics for each location:
bagging_cv_class = as.factor(bagging_cv$class)
levels(bagging_cv_class) = c(levels(bagging_cv_class), levels(actual))
bagging_cv_class = factor(bagging_cv_class, levels(bagging_cv_class)[order(
    levels(bagging_cv_class))])
confusion_bagging_cv = confusionMatrix(data = bagging_cv_class, reference =
    ACTUAL)
### Save Bagging results to machine
write.csv(as.data.frame.matrix(confusion_bagging$table), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    bagging_confusion.csv")
write.csv(as.data.frame.matrix(confusion_bagging$byClass), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    bagging_stats.csv")
write.table(error_bagging, file = "C:/Users/jdmunyon/Desktop/Dropbox/
    Senior_Project/data_and_code/results/50_50/bagging_stats.csv", append = TRUE
    )
### Save Bagging with ten-fold cross-validation results to machine
write.csv(as.data.frame.matrix(confusion_bagging_cv$table), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    bagging_cv_confusion.csv")
write.csv(as.data.frame.matrix(confusion_bagging_cv$byClass), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    bagging_cv_stats.csv")
```

```
write.table(error_bagging_cv, file = "C:/Users/jdmunyon/Desktop/Dropbox/
    Senior_Project/data_and_code/results/50_50/bagging_cv_stats.csv", append =
    TRUE)
### Save R workspace
save.image("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/50_50.RData")
```

SUPPORT VECTOR MACHINES:

```
library(e1071)
### "Refresh" seed - not really necessary:
set.seed(0)
### Read in ready-to-go data:
DATA = read.csv("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv", header = TRUE, row.names = 1)
### Set a seed for reproducability (since random sampling is coming up within
    the cross-validation):
set.seed(69)
### Determine optimal SVM parameters using ten-fold cross-validation
tune = tune.svm(x = DATA[,-1], y = DATA[,1], gamma = 2^(seq(from = -5, to = 15,
    by = 2)), cost = 2^(seq(from = -15, to = 3, by = 2)), tunecontrol = tune.
    control(sampling = "cross", cross = 10))
### Save the optimal parameters:
parameters = unlist(tune$best.parameters)
### Save a list of numbers, from 1 to the number of proteins in the dataset:
nums = 1:nrow(DATA)
### Create (3002) SVM models, each one using all but one left-out protein
### Then predict the left-out protein's location
### (this is jackknife validation):
svms = lapply(nums, function(z){
  a = svm(x = DATA[-z,-1], y = DATA[-z,1], cost = parameters["cost"], gamma =
      parameters["gamma"])
  b = predict(object = a, newdata = DATA[z,-1])
  return(prediction = b)
})
### Unlist svms (turn R object from a list to a vector):
svms = unlist(svms)
### Save actual locations of full dataset:
```

```
ACTUAL = DATA[,1]
### Calculate error rate of classification:
error_svms = 1 - sum(svms == ACTUAL)/nrow(DATA)
### Create confusion matrix and performance statistics for each location:
confusion_svms = confusionMatrix(data = svms, reference = ACTUAL)
### Save SVM results to machine
write.csv(as.data.frame.matrix(confusion_svms$table), file = "C:/Users/jdmunyon/
    Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/svms_confusion.
    csv")
write.csv(as.data.frame.matrix(confusion_svms$byClass), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    svms_stats.csv")
write.table(error_svms, file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project
    /data_and_code/results/50_50/svms_stats.csv", append = TRUE)
### Save R workspace
save.image("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/SVM_results_from_lab.RData")
```

COVARIANT DISCRIMINANT ALGORITHM:

```
### Read in ready-to-go data:
DATA = read.csv("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code
    /50_50.csv", header = TRUE, row.names = 1)
### Remove one amino acid from consideration:
DATA = DATA[,c(1,3:36)]
### Save character vector of locations:
classes = levels(DATA[,1])
### Split data into subsets, each one containing all proteins from one location:
subsets = lapply(classes, function(z){
  a = subset(DATA, DATA[,1] == z)
})
### Name each subset by its associated location:
names(subsets) = classes
### Create standard vector for each location:
standard = lapply(1:length(subsets), function(z){
  a = colSums(subsets[[z]][,-1])/nrow(subsets[[z]])
})
### Name each standard vector by its associated location:
```

```
names(standard) = classes
### Initialize (16) 34x34 covariance matrices, one for each location:
C = vector(mode = "list", length = length(classes))
C = lapply(C, function(z){
  return(matrix(0, nrow = 34, ncol = 34))
})
### For each location, fill-in the values of its covariance matrix:
for(z in 1:length(classes)){
  for(i in 1:34){
    for(j in 1:34){
      C[[z]][i,j] = (sum((subsets[[z]][,-1][,i] - as.vector(t(standard[[z]][i]))
          )*(subsets[[z]][,-1][,j] - as.vector(t(standard[[z]][j])))))/(nrow(
          subsets[[z]]) - 1)
    }
  }
  print(z)
}
### Name each covariance matrix by its associated location:
names(C) = classes
### Initialize (16) 34x34 inverse covariance matrices, one for each location:
C_inv = vector(mode = "list", length = length(classes))
C_inv = lapply(C_inv, function(z){
  return(matrix(0, nrow = 34, ncol = 34))
})
### Save each covariance matrix to machine (for later use in MATLAB):
for(i in 1:length(classes)){
  write.table(C[[i]], file = paste("C:/Users/jdmunyon/Desktop/Dropbox/
      Senior_Project/data_and_code/matrices_for_CDA/matrix", i, ".csv", sep =
      ""), sep = ",", row.names = FALSE, col.names = FALSE)
}
### Try to calculate inverses of covariance matrices in MATLAB ###
# M = csvread('c:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    matrices_for_CDA/matrix[i].csv'); % where [i] is in {1,16}, NOT in the
    brackets
# I = (p)inv(M); % this means try taking the inverse, and if can't, then take
    the pseudoinverse
```

```
# % although it ended up being the case that pseudoinverses didn't help either
# dlmwrite('c:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    matrices_for_CDA/inverse5.csv', I, 'precision', 10); % save covariance
    matrix
### End of MATLAB stuff ###
### Read in inverse covariance matrices:
for(i in 1:length(classes)){
  C_inv[[i]] = as.matrix(read.csv(paste("C:/Users/jdmunyon/Desktop/Dropbox/
      Senior_Project/data_and_code/matrices_for_CDA/inverse", i, ".csv", sep =
      ""), header = FALSE))
}
### Initialize (3002) length (16) vectors, one for each protein in the dataset:
MHs = vector(mode = "list", length = nrow(DATA))
MHs = lapply(MHs, function(z){
  return(vector(mode = "numeric", length = length(classes)))
})
### Name each entry in each MHs vector by its associated location:
for(i in 1:length(MHs)){
  names(MHs[[i]]) = classes
}
### For each protein, calculate the 16 similarity measures that go into its
    vector:
for(i in 1:length(MHs)){
  for(j in 1:length(classes)){
    MHs[[i]][j] = (unlist(DATA[i,-1] - standard[[j]]))%*%(C_inv[[j]])%*%(unlist(
        DATA[i,-1] - standard[[j]])) + log(det(C[[j]]))
  }
  print(i)
}
### Uh oh, for locations 5, 10, 11, 12, 13, and 15, even MATLAB couldn't
    calculate an inverse
### Pseudoinverses didn't help either, as (Matrix)*(Pseudoinverse) was not
    giving an identity matrix, or even close
### So for each protein, its similarity measure in location 5, 10, 11, 12, 13,
    and 15 is useless:
```

```
### Thus, to get around this problem, set them as the arbitrary large number
    10000000:
for(i in 1:length(MHs)){
  MHs[[i]][c(5,10:13,15)] = 10000000
}
### Create length (3002) character vector, each entry corresponding to a protein
predictions = vector(mode = "character", length = length(MHs))
### Determine location prediction for each protein by the location associated
    with its smallest similarity measure value:
for(i in 1:length(MHs)){
  predictions[i] = names(which.min(MHs[[i]]))
}
### Save actual locations of full dataset:
actual = DATA[,1]
### Calculate error rate of classification
### (note that we know all proteins actually in locations 5, 10, 11, 12, or 13
    will be predicted wrong):
error_CDA = 1 - sum(predictions == actual)/nrow(DATA)
save.image("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/CDA.RData")
### Note - will need to perform a "work-around" to ensure that the factor
### levels for the predictions are the same as those for the actual data.
### By default, will only have 10 levels, NOT 16, since 6 locations
### never get predicted for. Work-around is as follows:
### Save real predictions for first 6 proteins:
first_6 = predictions[1:6]
### Save 6 fake predictions, one for each location that never gets predicted:
fake_6 = levels(actual)[c(5,10:13,15)]
### Set predictions for first 6 proteins as the fake ones:
predictions[1:6] = fake_6
### Make predictions a factor - this will have the 16 needed levels:
predictions = as.factor(predictions)
### Put correct predictions back in for the first 6 proteins
### Number of levels will stay at 16 like we need!
### (Probably exists a better work-around: let me know!):
predictions[1:6] = first_6
```

```
### Create confusion matrix and performance statistics for each location:
confusion_CDA = confusionMatrix(data = predictions, reference = actual)
### Save CDA results to machine:
write.csv(as.data.frame.matrix(confusion_CDA$table), file = "C:/Users/jdmunyon/
    Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/CDA_confusion.csv
    ")
write.csv(as.data.frame.matrix(confusion_CDA$byClass), file = "C:/Users/jdmunyon
    /Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/CDA_stats.csv")
write.table(error_CDA, file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/results/50_50/CDA_stats.csv", append = TRUE)
### Save R workspace:
save.image("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/CDA.RData")
```

MIXTURE STRATEGY:

```
load("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50
    _50/MIXTURE.RData")
library(caret)
### Only need to consider numbers one less than each number of classes,
### and also each number of classes (so can see the increase/decrease each
    increment):
nums = c(nobs - 1, nobs)
### Sory by increasing:
nums = nums[order(nums)]
### Obtain predictions based on if/else statement
### Also, if location is one of the six that CDA fails on,
### use SVMs no matter what so there is a chance of good predictions:
predictions = lapply(nums, function(z){
  a = vector(mode = "character", length = 3002)
  for(i in 1:3002){
    if((nobs[which(ACTUAL[[i]] == names(nobs))] <= z) & (is.element(which(ACTUAL
        [[i]] == names(nobs)), c(5, 10, 11, 12, 13, 15))) == FALSE){
      a[i] = as.character(CDA[i])
    }
    else{a[i] = as.character(svms[i])}
  }
  return(a)
```

```
})
### Make predictions factors, not characters:
new_nums = 1:(2*length(nobs))
predictions = lapply(new_nums, function(z){
  return(as.factor(predictions[[z]]))
})
### SAME WORK-AROUND IDEA SEEN BEFORE TO ENSURE 16 FACTOR LEVELS:
predictions = lapply(new_nums, function(z){
  predictions[[z]] = as.character(predictions[[z]])
  ### Save real predictions for first 16 proteins:
  first_16 = predictions[[z]][1:16]
  ### Save 16 fake predictions, one for each location that never gets predicted:
  fake_16 = levels(ACTUAL)
  ### Set predictions for first 16 proteins as the fake ones:
  predictions[[z]][1:16] = fake_16
  ### Make predictions a factor - this will have the 16 needed levels:
  predictions[[z]] = as.factor(predictions[[z]])
  ### Put correct predictions back in for the first 16 proteins
  ### Number of levels will stay at 16 like we need!
  ### (Probably exists a better work-around: let me know!):
  predictions[[z]][1:16] = first_16
  return(predictions[[z]])
})
### Save confusion matrices and stats:
confusions = lapply(new_nums, function(z){
  return(confusionMatrix(data = predictions[[z]], reference = ACTUAL))
})
### Save classification accuracies:
accuracies = lapply(new_nums, function(z){
  return(confusions[[z]]$overall[1])
})
### Make a vector, not a list:
accuracies = unlist(accuracies)
### Give appropriate names:
names(accuracies) = as.character(nums)
### Scatterplot of nums and accuracies:
```

```
plot(nums, accuracies)
### Get indices of best (should be two of them):
indices = which(accuracies == max(accuracies))
### Best values and classification accuracies:
best = accuracies[indices]
### Take average of the two:
average = (as.numeric(names(best)[1]) + as.numeric(names(best)[2]))/2
### Save best confusion matrices and stats (are both the same):
best_confusions = list(confusions[[indices[1]]], confusions[[indices[2]]])
### Save workspace to machine:
save.image("C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/MIXTURE.RData")
### Save results to machine:
write.csv(as.data.frame.matrix(best_confusions[[1]]$table), file = "C:/Users/
    jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/results/50_50/
    mixture_confusion.csv")
write.csv(as.data.frame.matrix(round((best_confusions[[1]]$byClass), digits = 2)
    ), file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/data_and_code/
    results/50_50/mixture_stats_round.csv")
### Set c as a confusion matrix, previously calculated and stored
### (this confusion matrix has rows as predictions and columns as actuals):
c = as.matrix(as.data.frame.matrix(best_confusions[[1]]$table))
### Initialize length 16 MCC vector (note that the following code which creates
    MCC values if valid for any of the previous methods, assuming that the R
    object "c" has previously been saved as a confusion matrix):
MCC = vector(mode = "numeric", length = 16)
### For each location:
for(k in 1:16){
  A = as.double(c[k,k]) ### A = #(true positives)
  B = as.double(sum(c[k,]) - c[k,k]) ### B = #(false positives)
  C = as.double(sum(c[,k]) - c[k,k]) ### C = #(false negatives)
  D = as.double(sum(c) - A - B - C) ### D = #(true negatives)
  if((((A + B)*(A + C)*(D + B)*(D + C))^.5) == 0){ ### if denominator is zero,
      arbitrarily set as one - MCC will come out to be zero
    MCC[k] = (A*D) - (B*C)
  }
```

```
  else{MCC[k] = ((A*D) - (B*C))/(((A + B)*(A + C)*(D + B)*(D + C))^.5)} ### MCC
      value by formula
  rm(A,B,C,D)
}
### Rounding, setting as column vector:
MCC = round(matrix(MCC, nrow = 16, ncol = 1), digits = 2)
### Set appropriate column name:
colnames(MCC) = "MCC"
### Save results to machine (to be appended to pre-existing statistics by class
    table):
write.table(MCC, file = "C:/Users/jdmunyon/Desktop/Dropbox/Senior_Project/
    data_and_code/results/50_50/mixture_MCCs.csv", sep = ",", row.names = FALSE,
     col.names = TRUE)
rm(c, MCC)
```