

Humboldt Universität zu Berlin

Institute of Information Systems
Business Analytics and Data Science



Project Report


Group 8

Matriculation Number: 1234567
01.02.2017

Contents

1	Introduction	1
2	Background Information and Related Work	5
2.1	Tree Based Classifiers	5
2.2	Support Vector Machines	6
2.3	Logistic Regression	8
2.4	Oversampling	9
3	Data Pre-processing	11
4	Problem Description and Model Selection	13
5	Experiments	17
6	Conclusion	21

1 Introduction


Customer retention is a vital component of any modern retail business. It has been shown  **TODO: CITE ORIGINAL PAPER** that retaining existing customers is significantly cheaper than acquiring new ones and that the value of a customer for a company tends to increase dramatically over time. Some businesses accept that they will likely initially lose money on a new customer, i.e. will have to invest more than they will profit from their purchases, but this investment has been shown to pay off later. Customer churn is especially problematic for online businesses, where users are less likely to be attached to a certain company due to a lack of personal interaction with it or its employees and therefore more likely to leave. In order to prevent this, many companies rely on coupons to attract users to return. As these coupons forego much of the profits of the resulting purchase, it is important to target customers that wouldn't have returned otherwise while avoiding reducing profits from normally returning customers.




This work studies one such problem for an electronic retailer seeking to identify customers for a coupon campaign. Given a large set of "dirty" data, our task is to train a model that will generalize well to new customers and identify targets for the promotional campaign and obtain the best possible returns for the company. In the simple cost model presented, a correctly sent voucher results in 3 Euros of profit for the company. A "wasted" coupon, i.e. one that only discounts the purchase price to an already returning customer, directly reduces profits by its face value.

It is important to note that the problem formulation, which utilizes a single estimate for customer value, appears somewhat counter-intuitive. This is the case as all customers, regardless of their previous interactions with the company, are seen as having the same expected payoff in the future based on their future return status and eventual coupon they would receive. **TODO: VERIFY** As shown in our initial analysis, customers with larger orders are often more likely to return, which makes them a poor candidate for receiving a coupon, which contradicts the intuition that those customers are more valuable to the company due to them being more likely to make larger orders (with accompanying larger margins) again. This example shows that it is vital to consider the key model assumptions during every design decision, as relying solely on intuition during the design phase, e.g. focusing on retaining the big spenders, may hurt performance.

The task of building a prediction system for identifying valuable targets for coupons requires several components and deviations from typical models used for classification tasks presented in textbooks. The data is derived from a real world source so it is filled with numerous errors and inconsistencies which compromise its integrity and which need

to be addressed. There are nominal and cardinal features with significantly differing value ranges. While some models, such as those utilizing decision trees, can handle both cases at the same time, many others require the data to be processed or normalized to allow meaningful comparisons. It is also important to identify variables which are relevant for the classification task, i.e. provide meaningful insight into the problem, and removing those that don't contribute to the task. This is especially important when using metrics that rely on some sort of distance measure for learning, such as some kernel methods. By filtering out those variables, the resulting distance measures are far less noisy and easier to handle for the underlying model. This is closely connected to the issue of strong correlations among variables and their impact on performance. E.g. the weight of an order is derived from a combination of weights of the various items in the order. It is necessary to evaluate if the weight improves accuracy or merely contains a linear combination of item counts and average weights for the various categories in the order. Many of these questions can't be answered without examining the data in great detail and evaluating the impacts on the proposed models.

 Many methods seek to maximize accuracy or minimize the **mean square error of resulting predictions**. Our task's asymmetric cost structure requires a different approach. This needs to be reflected in the learning task, which is often disregarded in available software packages. There is no obvious best practice for this and it needs to be evaluated individually for each method considered. The best case scenario is having methods that directly incorporate this type of behavior into the underlying optimization problem of the classifier. This is the case for support vector machines (SVMs), which can handle different class weights using the LIBSVM package. We utilize libsvm's class weights to model the cost structure correctly show that the resulting decision function maximizes it. When this modification isn't possible it is necessary to adjust the data itself. We utilize a Oversampling increases the cost of training and runs the risk of skewing the data if done incorrectly. For this reason we utilize three different oversampling techniques.

 **A large number of training samples** makes it difficult to directly train some models. Kernel methods, which often exhibit excellent performance in various common machine learning problems  **DO: DO WE NEED SO MANY CLASSIFIERS** tend to scale  **proportionally with the size of the training data** **TODO: CITE ELEMENTS OF STATISTICAL LEARNING**. Traditional grid search and cross-validation techniques are infeasible for the large number of models that need to be trained to obtain a good classifier. We present our methodology for finding best performing hyperparameter ranges and combinations by efficiently allocating available processing time to models with varying sample sizes and utilizing information from the underperforming smaller models to speed up the parameter search.


We build our final model from a combination of our two best performing methods. We combine the best performing support vector machines with the best random forest classifier. The resulting method uses the prediction of random forest classifiers in areas


of high certainty i.e. for elements where the random forest classifier assigns very high probability of a customer belonging to a certain class. In areas where the random forest classifier is uncertain we rely on the support vector machine, which exhibited better performance in our validation phase. We explain why we opted for this method, explain other ensemble methods we evaluated and present the benefits of our mixed approach.


2 Background Information and Related Work

TODO: Introduce notation for x , y , N , M etc here as an intro to the chapter.


2.1 Tree Based Classifiers

Tree-based methods partition the feature space into a set of rectangles and fit a simple model (like a constant) in each one.  **TODO: THIS IS A DIRECT QUOTE FROM ELEMENTS.** Starting from the top node and working down, all points are evaluated along the nodes of the tree and passed on to the correct corresponding node. The resulting decision function can become very complex with a large depth of the tree, with various methods available for choosing the appropriate split criteria or leaf properties when training such trees. As decisions about splits are based on some distribution of training data that falls into that leaf node, decision trees can also output probabilities, thereby enabling further statistical analysis.

Individual trees are simple to use, can be visualized if necessary and can function well with data that hasn't been scaled, extended with dummy variables and even with missing values.  **DO: CITE SCIKIT.** The complexity of training a tree is logarithmic in the number of data samples which is orders of magnitude faster than many other popular methods (such as SVMs). They also support our initial hypothesis that the classification problem studied in this work has at least some customer profiles that exhibit very similar return habits while sharing few important properties (e.g. large orders during the holidays). Methods that take all attributes into account might overlook such trends due to noise introduced by other attributes in such cases, while decision trees are likely to place all clear groups in some leaf node.

The listed properties are evened out by some significant problems that need to be overcome. Even if a "perfect" tree might exist, i.e. one that performs a 100% accurate classification, the task of learning the optimal tree is NP-complete and requires various heuristics for choosing and evaluating splits within the tree. Very deep trees can easily overfit and are sensitive to the presented data.  **TODO: CITE ELEMENTS AND SCIKIT** Because the problem studied in this work is complex with a large number of features, poor generalization is a very real threat when utilizing a single tree.


Although the listed disadvantages make using a single decision tree a poor option,

they have been shown to function well in various types of ensembles, serving as the ideal "weak learner" described in various boosting methods. The simplest ensemble method is bagging, where numerous trees are trained on parts of the training data and the final prediction takes all trees into account. This resulting estimator should then have lower variance than the high-variance trees it uses. It is possible that certain features will be seen as important in every sample and serve as split criteria at the top of most trees, thereby making the learned trees correlated. This in turn reduces the benefits of bagging and the associated reduction in variance  **TODO: CITE ELEMENTS.**

Random forest classifier address this weakness by utilizing bagging of data samples and the features themselves. Only a *random sample of the features* is used when determining each split criterion **TODO: CITE SANDRAS ELEMENTS.** The resulting trees are decorrelated. It is necessary to determine the number of features to be selected for each split. According to **TODO: CITE ELEMENTS AND SANDRAS ELEMENTS** the rule of thumb is taking the square root of the number of features.

A different approach for building ensembles using decision trees is boosting. Rather than sampling randomly, boosting utilizes an iterative approach that attempts to minimize a loss function and improve the resulting *adaptive model* in each iteration. The initial weak learners don't change, but subsequent learners are trained to optimize the given function in unison with existing learners. The popular AdaBoost method works on residuals, i.e. current deviations of resulting predictions from the base model, rather than training each classifier for the true labels. *Gradient boosting* extends the concept to differentiable loss functions (not just squared error) and trains weak learners that point in the negative direction of the gradient, acting much like gradient descent.

2.2 Support Vector Machines

Support vector machines are a type of linear classifier that finds a separating hyperplane between two classes, such that the the distance of the closest element of each class is  ximized. This principle, called margin maximization, was shown in **TODO: CITE VAPNIK** to improve generalization in case of linearly separable data. Formally, SVMs find a vector w that solves the following problem for a set of vectors (x_1, x_2, \dots, x_N) and their corresponding labels (y_1, y_2, \dots, y_N) :

TODO: FIGURE OF MAX MARGIN HYPERPLANE FROM VAPNIK OR SCHOLKOPF. SIDE BY SIDE WITH MAX MARGIN

$$\begin{aligned} \min_w \quad & \frac{1}{2} \|w\|^2 \\ \text{s.t.} \quad & y_i(w^T x + b) \geq 1 \end{aligned} \tag{2.1}$$

Minimizing the norm of w under these constraints corresponds to maximizing the

margin, shown in TODO: A FIGURE AND REFERENCE IT. While this convex optimization problem is solvable (for linearly separable data), it is often more convenient to solve the *dual problem*. This is done by looking at FOC and substituting them into the Lagrangian. The dual problem then becomes:

$$\begin{aligned}
 & \max_{\alpha} \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j x_i^T x_j \\
 & s.t. \quad \sum_{i=1}^N \alpha_i y_i = 0 \\
 & \quad \alpha_i \geq 0, \quad \forall i \in \{1, \dots, N\}
 \end{aligned} \tag{2.2}$$

with the vector w then being $w = \sum_{i=1}^N x_i \alpha_i$ and be being calculated by solving $w^T x_k + b = 1$ for any k s.t. $\alpha_k > 0$. This is commonly known as the *hard margin SVM*. An alternative, far more popular approach is the so-called *soft margin SVM*, which allows for some variables to be inside the margin or even be assigned to the wrong class, providing smoother and better regularized decision surfaces as a result. The resulting problem is the same, with an additional constraint of $\alpha \leq C$ where C determines how "expensive" deviations in the model are. The name of the classifier is derived from the fact that the resulting hyperplane is determined as a linear combination of data points from the training set, known as *support vectors*. In this simple case those vectors are the ones TODO: FIND A PIC AND DEFINE THEM, with $w^T x + b = \pm 1$. While this property is nice, it's not really necessary for the linear classifier unless the number of features in the data exceeds the number of training points. The biggest advantage of the dualized form of the problem is in the use of complex transformations of the data, while maintaining the process simple and mapping obscured through the use of the *kernel trick*. The *mapping* Φ to a *feature space* F which transforms the data. If a good transformation is chosen, even complex non-linear problems can become linearly separable. Formally we have:

$$\begin{aligned}
 \Phi : \mathbb{R}^N &\rightarrow F \\
 x &\rightarrow \Phi(x)
 \end{aligned} \tag{2.3}$$

which inserted into the SVM problem then gives:

$$\begin{aligned}
 & \max_{\alpha} \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \Phi(x_i)^T \Phi(x_j) \\
 & s.t. \quad \sum_{i=1}^N \alpha_i y_i = 0 \\
 & \quad \alpha_i \geq 0, \quad \forall i \in \{1, \dots, N\}
 \end{aligned} \tag{2.4}$$

Kernel Name	$k(x, x')$
Linear	$x^T x' + c$
Polynomial	$(ax^T x' + c)^d$
Gaussian	$\exp(-\gamma \ x - x'\ _2^2)$
Sigmoid	$\tanh(ax^T x' + c)$

Table 2.1: List of popular kernel functions

The classifier then assigns a label to a new data point based on:

$$y = \text{sign}\left(\sum_{i=1}^N \alpha_i \Phi(x_i)^T \Phi(x) + b\right) \quad (2.5)$$

The *kernel trick* takes advantage of the fact that the whole process of scoring new points doesn't need w explicitly computed, which can become very costly if the feature space is high dimensional. Instead, it is sufficient to have a defined *kernel function* that calculates the inner product within the feature space i.e.

TODO: FIGURE FOR THE KERNEL TRICK, BEST TO USE THE ORIGINAL FROM VAPNIK OR MUELLER ET AL

$$k(x_i, x_j) = \Phi(x_i)^T \Phi(x_j) \quad (2.6)$$

An added benefit of the model is that classification only requires computing the kernel function between a new point and the support vectors, as all other points have $\alpha = 0$ and therefore don't contribute to the score. The final classifier then only requires α and the support vector for classifying new points. While the optimization problem requires a costly quadratic programming solver to be used, the resulting set of support vectors can be very small compared to the size of the data set and provide fairly fast classification.

2.3 Logistic Regression

Logistic regression is a linear model for classification which can be seen as an extension of linear regression to classification TODO: CITE ELEMENTS. In its simplest form for binary classification, it calculates the probability of $p(x)$ of a point belonging to a certain class. The probability of it belonging to the remaining class is then $1 - p(x)$. The probability is calculated by using the *logistic* function, from which it derives its name. For a given vector w and offset b the probability is then $p(x) = \frac{1}{1 + \exp(-(w^T x + b))}$. The coefficients of w and the offset are found by solving TODO: CITE SCIKIT:

$$\min_{w, b} \sum_{i=1}^N \log(\exp(-y_i(w^T x_i + b)) + 1) \quad (2.7)$$

The model can be regularized by contributing the L1 ($\|w\|_1$) or L2 ($\frac{1}{2}w^T w$). Contrary to what we usually observed in regularization of other methods, the scaling parameter C which governs the importance of regularization in the cost function is not used to scale the regularization term, instead having the form (e.g. for L2 regularized logistic regression):

$$\min_{w,b} \frac{1}{2}w^T w + C \sum_{i=1}^N \log(\exp(-y_i(w^T x_i + b)) + 1) \quad (2.8)$$

Variations between logistic regression modules of various tools tend to only differ on the available solvers for the optimization problem, with no other variables being provided.

2.4 Oversampling

Oversampling attempts to adjust classifier performance to account for uneven class distributions and asymmetric cost structures while still maintaining the same optimization problem and corresponding solver. It is useful when modifying the problem itself isn't possible. The simplest form is oversampling with replacement, where additional samples from the minority class are drawn with replacement until the desired ratio of classes is achieved. As noted in TODO: CITE CHAWLA, it has been observed that oversampling with replacement doesn't significantly improve minority class recognition.

A method for oversampling by generating synthetic examples from the minority class is SMOTE (Synthetic Minority Over-sampling TEchnique). A new sample is generated from a data point x_i , one of its k nearest neighbors x_j and a factor α such that the new points $\hat{x} = \alpha x_i + (1 - \alpha)x_j$. i.e. the sample is placed between the point and its selected neighbor. This in turn "fills-out" the area between the minority class samples, changing the importance of the minority class in the model and hopefully also leading to better decision surfaces. The method can be further enhanced by mixing it with an *undersampling* technique identifying and removing Tomek links, which are pairs of elements belonging to different classes that are each other's closest neighbors. This serves to filter out potentially noisy samples of the majority class mixed with the minority class.

TODO: SOME SIDE BY SIDE COMPARISON OF THE METHODS, COUL BE NICE

2.5 PCA

TODO: This is pretty much optional. Can be included because PCA was used (afiak it didn't help) and kernel PCA is interesting to visualize when talking about the kernels, but it isn't a must. Gives us nice plots if we need more space.

3 Data Pre-processing

Work in progress. Will combine old version with Tim's updates.

4 Problem Description and Model Selection

The task of this project is identifying targets for dispatching coupons in the company's effort to retain customers. We are given a large set of records on previous customers and required to identify a list of targets for the coupon campaign. To identify targets for the promotional campaign, we divide the work into the following stages:

- Define a metric or set of metrics by which proposed methods can be compared
- Choose a group of classification methods to be used
- Find a method of incorporating the asymmetric cost structure into the model, either through oversampling or modifying the classifier itself
- Select a technique for cross-validation
- Run the planned experiments and pinpoint best performing models
- Identify potential improvements and determine final predictions

As the goal of the company is maximizing profits, it is clear that the primary performance metric should be based on the expected profit. As we wish to compare profits between different sample sizes and splits it is more convenient to calculate expected profits per customer. We mark this quantity as ROI (return of investment), throughout the code and paper. While this measure normally takes the initial investment into account, we disregard it and the entire denominator as our "investment" has no costs. The resulting metric is then computed as:

$$ROI(\hat{y}) = \frac{1}{N}(3 * TN - 10 * FN) \quad (4.1)$$

where TN and FN are the numbers of true negatives and false negatives respectively, with a negative in our case being a person that would not return.

In addition, it is helpful to look at the impact of the threshold used in models that assign probabilities to classes and its impact on the profits. We use this instead of looking at ROC curves. Not wishing to discard them completely, we also use the area under curve measure when possible in case we need tie-breakers. For SVMs, we mimic the behavior of varying the threshold by checking how a *bias introduced into the classification function itself changes performance*. When evaluating the model we introduce a bias term t into the

decision function such that classes are computed as $y(x) = \text{sign}(\sum_{i=1}^N \alpha_i k(x_i, x) + b + t)$, effectively moving the separating hyperplane. This serves as a sanity check for the resulting SVM, where a noticeable increase in profits by picking a t that isn't close to zero signifies an issue with the learned classifier. For a classifier that already exhibits poor performance this is simply another system of poor parameter choices. If the classifier is showing good performance but has this property, it can signify room for improvement by changing the C parameter, or indicate a poor training/validation split which ought to be changed if the problem persists for other parameter combinations. Examples of this are provided in the TODO: REF NEXT CHAPTER

Our first choice of classifiers were decision tree based classifiers, namely random forests and gradient boosting. They are known to perform well TODO: CITE DO WE NEED HUNDREDS OF CLASSIFIERS, can work well with mixed data types, are resistant to poor scaling of variables and due to their various inbuilt regularization techniques are likely more forgiving to possible mistakes in the data processing phase. They also have the benefit of having the most understandable underlying method for generating predictions and are the most intuitive among the studied methods.

Logistic regression was included due to its popularity in various sources we encountered. In addition, good performance here could indicate potential applicability of other linear methods to the problem.

Support vector machines were not originally planned due to the high training cost and need for hyperparameter. Our initial plan was to use neural networks due to the recent surge in their popularity due to deep learning and advanced training methods utilizing GPUs. They proved to be far more difficult to understand and train using sklearn than other evaluated methods, with massive issues with convergence and non-convexity during our prototyping. We are certain that our chosen model can be outperformed by a well tunes multilayer perceptron, but unlinke with SVMs there was no obvious route to take for dealing with these issues.

Our design was based initially only relied for k-fold cross validation. This method splits the data into even partitions and performs k different runs, with each partition being used for validation once. As it is necessary to tune the various parameters of the model and then obtain an unbiased estimate on model performance, our desired way of doing this called for nested k-fold cross-validation, i.e. having an outer loop that goes over test sets and an inner loop where model selection and tuning is conducted. This was too resource intensive for all of the proposed benefits and the additional cost of the outer loop was not justified by the benefits it provides. The procedure used for training most methods (everything except SVMs) then relied on the following:

- Split the data into a two large sets with 85% and 15% of the data respectively.
- Using the large set, perform cross validation and model selection using k-fold cross

validation. We used $k=4$ for more complex methods and $k=8$ for others.

- Once all the models have been trained and the best one identified, use this model on the test set and evaluate performance. This is the final estimator for our model. The decision isn't made based on the test set, the test set only serves as an estimate.
- Train the final model with the entire data using the parameters we found

TODO: IF POSSIBLE REWRITE THIS AS PSEUDOCODE. (I don't think it will actually be good as it will be longer and more complicated).

This method proved a poor fit for evaluating support vector machines which experienced a very large increase in computational cost when going from 25000 to 40000+ training samples. This was especially true for high values of the parameter C , where the machine acted more like a "hard margin" SVM and occasionally took three to five times as long to train than with smaller values. Instead of reducing the number of folds and therefore the number of runs used for validation, we opted instead to use the same number of runs using a smaller number of training points and a different sampling method. We opted for *Monte Carlo cross-validation*, where for a given number of runs (in our case 5) the data is randomly partitioned into a training and set sample. To make sure the runs are comparable, we used a seed for the pseudorandom number generator, making the tests repeatable. This difference in approach resulted in significantly less expensive validation procedure, enabling more runs and therefore more effective coverage of the interesting parameter ranges.

TODO: Figure from the weather book with the nice plots for the different validation procedures.

5 Experiments

Work in this project was divided into two sections utilizing different tools. Data processing, cleaning and feature extraction were done in the R environment. TODO: MORE ON PREPROCESSING. Most of the work for model selection, hyperparameter evaluation and training was done using the scikit-learn package for Python, along with NumPy, SciPy and matplotlib packages on top of which it is built. The bulk of SVM training was done in the libsvm package, which the sklearn.svm package serves as a wrapper for. Pandas was used for handling data frames within Python. The imbalanced-learn package was also used, which is a branch of sklearn.

All training and cross validation for random forests, gradient boosting and logistic regression was done using scikit learn and its associated methods. All the classifiers follow the same layout structure and encoding of parameters, enabling large parts of the code for cross-validation and parameter selection to be reused, which is one of the reasons we opted for sklearn. In addition, due to the embarrassingly parallel nature of random forest training, i.e. training tasks for various trees being completely independent of each other, we were able to utilize sklearn's inbuilt parallelization capabilities. The resulting code could efficiently utilize all cores during training significantly speeding up the process.

TODO: MORE ON DAVID'S RANDOM FOREST TRAINING

SVM training was far more difficult due to the very high cost of training individual SVMs on large data sets. With no clear idea what valid parameter ranges for parameters could be, it was necessary to evaluate a parameter space using a grid search and gradually narrow down the valid parameter ranges. As initial attempts to utilize sklearn's parameter search capabilities repeatedly stalled and caused system instability, we opted to perform the search manually by observing previous results and gradually narrowing the field. To avoid accidentally eliminating important portions of the parameter space, we also included areas close to the best performing combinations in subsequent runs. TODO: Give plots and specific examples.

According to TODO: CITE SCIKIT, complexity of the underlying quadratic programming solver used by libsvm for training individual models ranges between $O(MN^2)$ and $O(MN^3)$ where M is the number of features and N the number of training samples. We observed very large variations in our testing, with a worst-case increase in training time by a factor of 25 when doubling the training sample size from 15000 to 30000. At the same time, we observed a noticeably smaller increase in memory use (800MB to 1.1GB) than expected, signifying that the culprit may be the libsvm cache size mentioned in

sklearn's documentation. As the utilized ipython notebook was capable of drawing more system memory, we assume this was caused by some inbuilt libsvm property which can't be altered directly via sklearn. Admittedly, the largest models we used significantly exceeded the recommended sizes TODO: CITE SCIKIT.

As all solvers hovered around 30-35% CPU utilization on a quad-core processor, we increased the number of threads. The initial attempt of using the parallel python package resulted in the Windows 10 interpreting the behavior as potentially malicious and devoting most of the CPU time to its inbuilt Windows defender program, at times even reducing the CPU utilization for SVM training below the original 30%. The problem was solved by normally starting several ipython kernels and running training independently.

It quickly became clear that polynomial kernels were underperforming when compared to gaussian kernels. The best large SVM trained using a polynomial kernel exhibited ROI around 0.7 and was therefore quickly scraped due to obvious inferiority compared to random forests and the more promising gaussian kernels. All the effort (and more importantly processing power) for SVMs was then focused on gaussian kernels.

Performance of SVMs using gaussian kernels is determined by parameters γ and C . In addition, it is not possible to pinpoint the optimal value for either independently, as shown in figure TODO: INSERT FIGURE HERE. From demos TODO: CITE SKLEARN and tutorials on SVMs it was clear that the parameter γ , which regulates the 'neighborhood' size or rather the decay factor for the distance at which data points contribute to the score, is the crucial element for performance. When too large, the resulting classifier takes very large neighborhoods into account and results in an oversimplified decision function. If selected to be too small, the classifier is almost certain to overfit and perform poorly on all variables that aren't very similar to one of the training samples.

SVM performance depends heavily on the choice of parameter C , which regulates the cost of missclassification and deviations from the "hard margin" SVM. Values of C that are too small are likely to result in oversimplified models. Increasing C then leads to a better fit and generalization as the outliers and values within the margin become more expensive in the cost function. Eventually increasing C past a certain point makes the problem more computationally expensive while contributing little to the end solution. We verified this assumption during training by comparing performance of several models using very large values of C $\{10^3, 10^5\}$ on various values of gamma, which resulted in SVMs having nearly identical ROI and number of support vectors as elements with smaller C values, but with training times occasionally being three times longer.

TODO: Explain parameter selection procedure for SVMs, how the best performing combination changes with the sample size (larger sample -> works better with lower C and gamma) and how best performing models always have a relatively high number

Table 5.1: My caption

Method	Observed ROI on the test set
Random Forests	(NOT ACTUAL NUMBERS) 0.74
Random Forests with Oversampling	0.78
Random Forests with SMOTE	0.8
Random Forests with SMOTE + Tomek	0.82
Gradient Boosting	0.81
SVM	0.95
Logistic Regression	0.6
etc	etc

of support vectors. Show plots for the training procedure, side by side to save space. Explain the thresholds and their function.

TODO: EXPLAIN THE SMALLER BOOTSTRAP SVM MODEL AND WHY IT WAS DUMB.

TODO: MUCH OF DAVID'S PART STILL LEFT

TODO: Nicer figures showing the following: Stages of SVM training, The random forest vs SVM compromise, the plot showing how and where they differ,

TODO: Explain the mixed model, why we opted for it instead of trying to build an ensemble using proper classifiers and this impacted the design process.

6 Conclusion

TODO: Short overview of what was done, the specific of the proposed model, the number of identified return customers and possible improvements. Finished last.

