Multi-Objective Multi-Task Learning

Abraham Bagherjeiran

Draft: $Revision: 764 $ May 10, 2007

This dissertation presents multi-objective multi-task learning, a new learning framework. Given a fixed sequence of tasks, the learned hypothesis space must minimize multiple objectives. Since these objectives are often in conflict, we cannot find a single best solution, so we analyze a set of solutions. We first propose and analyze a new learning principle, empirically efficient learning. From a sample complexity perspective, following this principle is not much worse than the single-objective multi-task learning case. In the context of empirically efficient learning, algorithms for the new learning frameworks are proposed and evaluated. First, we pose regularization as a multi-objective problem, in which training error must balance the complexity of the hypothesis space. Second, we consider multiple data-dependent loss functions, in which the error rate in one class must balance the error rate in the other class. Finally, we assume that tasks share a clustering structure in which the average loss in one cluster must balance the loss in another cluster. The algorithms are evaluated on synthetic and real datasets. The results motivate the application of multi-objective optimization, indicating that the objectives are in conflict. By controlling the relative performance of the algorithms to generate a tradeoff surface, we can effectively explore the multi-objective nature of the learning problem.

Contents

1 Introduction

In contrast to single-task learning, multi-task learning enables us to learn from similar learning tasks, offering several key benefits. First, knowledge can be transferred across learning tasks, converting the single-task learning problem into one that is more informative and that helps narrow the set of potential solutions. Second, recent work has established a theory of the sample complexity of multi-task learning, in which a global loss function is minimized across all datasets. A key result is that the availability of similar learning tasks simplifies the learning problem; formally, it has been shown that the average number of examples needed per task decreases as the number of tasks increases (under a fixed error ϵ [13]). The corresponding bounds have been further improved using the Rademacher complexity applied to multi-task learning [58].

A key disadvantage of multi-task learning, however, is the assumption that we want to minimize a single global loss function. In applications such as meta-learning, there can be multiple objectives such as recommending a learner that is both the most accurate and fastest. Often these objectives are in conflict, a fast learner may not be accurate and an accurate learner may not be fast. Thus, it may be the case that there is no learner that is both accurate and fast. The best we can do is find a compromise, which we call an efficient solution. We may find several efficient solutions, so that a learner may be the most accurate for its speed. The learning algorithm should return this set so that we can analyze the tradeoffs, how much slower is a more accurate algorithm, which we cannot do with existing methods because they are only interested in a single solution. An algorithm determining that accuracy is 2.77 times as important as speed does not tell us as much as an algorithm discovering that for every 1s improvement in speed we lose 10% in accuracy. The main contribution of this work is to show that not only can we design multi-task algorithms that learn this set of efficient solutions but that the proposed framework has a consistent and solid theoretical foundation.

Regularization algorithms, such as support vector machines, balance the complexity of the learned solution against the training loss [76]. This common approach has been recently extended to multi-task learning where the complexity of the hypothesis space is balanced against the training loss across all tasks [33, 77]. Although the objectives are in conflict, there is really only one data-dependent objective, the loss on the training set. The other depends on the size of the hypothesis space and not its output. Thus, we say that it is data-independent. Regularization methods require a user-defined parameter to manually balance the conflicting objectives, whose value is unintuitive and often requires searching over the parameter space. The most common method used is cross-validation, but as the number of objectives increases, the size of the search space of parameters increases exponentially making the approach impractical [47].

In applications such as fraud detection, we want to control the false positive and false negative rates over all tasks. Specialized algorithms and methods have been developed in the field of pattern classification, such as ROC curves and the F-measure [34]. These methods, however, have only been applied to single-task learning, and it is unclear how to define a ROC curve for multi-task learning. The threshold for the class-conditional distribution would have to be selected for each learning task separately after the algorithm is applied to a test set. In addition, these methods are only suited to a specific set of loss functions. If there are domain-specific loss functions defined on a per-example basis rather than a per-class basis, such as lost revenue for making bad recommendations, the ROC approach is not applicable. In contrast, the expected set learned by our proposed algorithms is defined for any objectives and does not require finding a threshold for each task. Instead, each solution in the efficient set is a different learned function whose expected objective values are guaranteed to converge to their expected values.

A common method of incorporating multiple objectives is to learn a scalar rank functions, which is

Symbol	Description
k	Number of objectives
m	Number of tasks (datasets)
n	Number of examples in each dataset
q	Number of attributes in each example
i	Objective index $1 \leq i \leq k$
t	Task index $1 \leq t \leq m$
j	Example index $1 \leq j \leq n$
r	Attribute index $1 \leq r \leq q$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
T_t	Learning task, a distribution on $\mathcal{X} \times \mathcal{Y}$
X^t	Dataset sampled from task T_t
$(\mathcal{X} \times \mathcal{Y})^{mn}$	Set of all multi-task datasets: m datasets with n examples
x_j^t	An example from dataset X^t
Φ	Task hyper-distribution
$\sim$	Distributed according to some probability distribution
h	Hypothesis $h : \mathcal{X} \to \mathcal{Y}$
$\mathcal{H}$	Hypothesis space
$\mathbb{H}$	Hypothesis family
l	A loss function $l : \mathcal{Y} \times \mathcal{Y} \to [0,1]^k$
l_i	ith component of the loss function
$L(h, X)$	Average loss on a dataset: $\frac{1}{n}\sum_j l(h(x_j), y_j)$
$L(h, T_t)$	Expected loss on a task: $\mathrm{E}_{(x,y)\sim T_t}\left[l(h(x), y)\right]$
$L(\mathcal{H}, X)$	Average loss on a multi-task dataset: $\frac{1}{m}\sum_t L(h_t, X_t)$
$L(\mathcal{H}, T)$	Expected loss on a multi-task sequence: $\frac{1}{m}\sum_t L(h_t, T_t)$
$\hat{\Theta}_E$	Empirical efficient set
$\hat{L}_N$	Empirical tradeoff surface (undominated set)
Θ_E	Expected efficient set
L_N	Expected tradeoff surface (undominated set)
λ	Vector of weights for the objectives, $\lambda \in \mathbb{R}^k$, $\lambda_i \geq 0$
w	Weight vector applied to an example, $w \in \mathbb{R}^q$

Table 1.1: Notation used in this dissertation.

Description	Relationship
$a \sim P$	a distributed according to distribution P
Hypothesis spaces	$h \in \mathcal{H} \in \mathbb{H}$
Examples in task	$(x, y) \sim T_t$
Examples in dataset	$(x_j^t, y_j^t) \in X^t$ for $1 \leq j \leq m$ and $1 \leq t \leq n$
Task sequence $T = \{T_t\}$	$T_t \sim \Phi$ for $1 \leq t \leq n$
Vector-valued loss function	$l(y, y) = (l_1(y, y), \ldots, l_k(y, y))$
Class labels	Given $(x, y) \in X$, $cl(x) = y$

Table 1.2: Important relationships assumed in the text.

commonly done in meta-learning. Similar to information retrieval, a user provides feedback indicating a preference for several learners matched to a particular dataset. The desired rank of the learner in the results is then learned [20, 51]. This method fails if a user is not available or qualified to interactively learn the weights. We also cannot analyze the tradeoffs between the different objectives because the scalar function results in a single solution in the efficient set. In our proposed learning frameworks, we can analyze the space of possible scalar functions all at once without having to re-run the algorithm for each possible scalar function.

Many multi-objective optimization methods have been proposed, but none has been applied to the problem of multi-task learning [27, 29]. We use multi-objective optimization algorithms, but our focus is on learning. The key difference is that we are not only interested in solving the multi-objective problem. The learning problem essentially starts only after the optimization is complete, where we must then determine whether the learned solution generalizes to new problems. This is in contrast to optimization in which the algorithm is concerned only with finding a solution and not how well the solution generalizes to new examples.

In learning the tradeoffs between objectives, we want to know how far the efficient set obtained from a finite sample is to the true efficient set given full knowledge of the data. In quantifying the convergence between the empirical and expected efficient sets, none of the existing multi-task theory can be directly applied. Here, we have generalized the existing theory to handle multiple objectives. The surprising conclusion is that attempting to learn the efficient set increases the number of examples needed per task only linearly with respect to the number of objectives.

This dissertation presents a theoretical and algorithmic framework for multi-objective multi-task learning. Drawing on results and algorithms from multi-objective optimization, a multi-objective multi-task algorithm learns a set of efficient hypothesis spaces given a fixed sequence of datasets [29]. Three new frameworks for multi-objective multi-task learning are proposed to address many of the limitations of the frameworks. In regularization, we can select one element of the efficient set, by adopting a cross-validation approach. In contrast, our multi-objective regularization framework outputs the efficient set directly, significantly reducing the search space. In the multiple-loss function framework, we can obtain the effect of a multi-task ROC curve without having to define a threshold for each learning task. The result is a curve much like the ROC curve but in which each point corresponds to a different hypothesis space rather than a threshold for each task. In the task clustering framework, the average loss for a cluster of tasks is balanced against the others. Evaluating our loss functions over the entire efficient set gives us a new set, called the tradeoff surface. This allows us to analyze the tradeoffs between the learned hypothesis spaces. From this tradeoff surface, we can easily find or learn a scalar rank function, which is simply a point on the tradeoff surface.

1.1 Goals

The main goal of this work is to incorporate multiple objectives into a multi-task learning algorithm. It is worthwhile to do multi-objective multi-task learning because many applications require optimizing several conflicting objectives, which is beyond the capabilities of existing multi-task learning algorithms. This is particularly important in recommender systems. In predicting what products a consumer wants to purchase, we can view each consumer as a learning task. Through multi-task learning, we can gain experience for a new consumer by learning on the large number of prior consumers. The goal of applying the learning algorithm is to recommend products that a consumer is likely to buy. We therefore consider the consequences of the recommending a product, which existing multi-task learners cannot accommodate. Different consumers, or groups of consumers, could respond differently to different recommendations. Different products could have different costs associated with them making them more or less profitable to recommend. Since these are often conflicting objectives, we want to recommend products to consumers when we are reasonably confident they want the product. On the other hand, products with a higher profit margin should be more

aggressively recommended even if a consumer may not prefer to buy them. Existing algorithms simply cannot handle these complex problems, often requiring the practitioner to tweak parameters or adopt heuristics that work some of the time. Multi-objective optimization allows us to consider the tradeoffs between these objectives, but there is currently little work in bringing this information into a learning algorithm. By adding this ability to multi-task learning, we have extended the frontier of what is available for learning algorithms.

1.2 Contributions

This dissertation contains several novel contributions to the existing body of knowledge. The primary contribution is the incorporation of multiple objectives into the existing multi-task learning framework, both theoretically and in the algorithms. As a byproduct of this work, a new open source software library was created and made publicly available. The main contributions are:

- Empirically Efficient Learning Frameworks
  - Regularization
  - Multiple Loss Functions
  - Task Clusters
- Sample Complexity Bounds
  - One-sided bounds for weighted objectives
  - One-sided bounds for multiple loss functions
  - Two-sided bounds for multiple loss functions
- Algorithms
  - Extensions to a multi-task SVM algorithm to handle multiple objectives
  - Multi-Task Multi-Objective distance function learning algorithm
- Software
  - Cougar^2: an open-source library for machine learning and data mining

The empirically efficient learning principle requires that we return an efficient solution with respect to a finite sample. By efficient we mean that the loss should be Pareto-optimal, such that all other points are no better in all objectives. The principle formalizes the solution concepts for a multi-objective multi-task learning algorithm, the empirical and expected efficient set. The consistency of the learning principle is derived in Chapter 2. Within this principle, three different classes of algorithms are proposed. In regularization algorithms, we balance the complexity of the solution against a single loss function defined on the training set. For multiple loss functions, we directly can compare the expected and empirical tradeoff surface. Finally, if tasks belong to a set of clusters, we can balance the loss in one cluster against the loss in another.

The empirically efficient learning principle is shown to be consistent and to generalize the sample complexity results of the original single-objective formulation of multi-task learning. Specifically, one-sided bounds are established for composite and vector-valued loss functions, showing that the sample complexity increases only linearly with the number of the objectives. We then show that this framework generalizes the existing single-task and multi-task learning theory. Next, two-sided bounds are derived for the multiple loss function framework, again generalizing the existing results.

In contrast to other theoretical works in multi-task learning, we apply our theoretical results to design four new multi-objective multi-task algorithms. An existing algorithm is extended to handle different types of conflicting objective functions. A new multi-task distance function learning algorithm is proposed that applies convex optimization to learning weights for a distance function. All algorithms are then experimentally evaluated on artificial and real datasets. The experimental results show that the algorithms can learn an efficient set of solutions that represent different hypothesis spaces.

Finally, existing software packages were neither designed for nor easily extended to multi-task learning [64, 79]. As a result, we created a new software library. The library has been released to the community as an open-source project. It includes the proposed algorithms and provides significant flexibility in designing new algorithms.

1.3 Road map

This dissertation is organized as follows. The remainder of this chapter presents some necessary background material in multi-objective optimization and machine learning. It ends by proposing our new multi-objective multi-task frameworks. Chapter 2 examines the sample complexity of the frameworks. The main results are that the number of examples needed per task increases linearly with the number of objectives and that the bounds generalize existing bounds. Chapter 3 proposes algorithms for the multi-objective multi-task learning framework and provides some implementation details. Chapter 4 evaluates the algorithms on artificial and real-world multi-task datasets. Chapter 5 discusses related work in multi-task learning, multi-objective learning, and meta-learning. Chapter 6 details the design of the software framework used to implement the experiments.

1.4 Example

As an example, assume we want to be able to predict whether any consumer from a population likes or dislikes a product. We ask several consumers to fill out a survey with a list of 20 randomly selected products, indicating whether he or she likes or dislikes a product, x. After the survey, we associate with each product x a consumer's preference, $y \in \{L, D\}$, either liking (L) or disliking (D) the product. Our goals are to predict whether:

Single-Task One specific consumer prefers a product.

Multi-Task Any consumer (from a certain population) prefers a product.

Multi-Objective Any consumer (from a certain population) prefers a product, and:

1. Minimize the cost of sending advertisements for products.
2. Learn a group of "premium" consumers better than "regular" consumers.

Single-Task In single-task learning, we can predict whether one particular consumer likes any product. We find a hypothesis, a function $h^c(x) \in \{L, D\}$ mapping each product x to consumer c's preference. Since this function is specific to one consumer, we cannot expect it to work for another consumer c'. We will have to start all over for each consumer, asking the same 20 questions, learning a completely new model each time.

Multi-Task In multi-task learning, we transfer our experience from one consumer to another because we expect that many consumers have similar preferences. Ideally, we want to learn only what is different between the consumers. Although we still learn a model for each consumer, it now has two parts: $h^c(x) = \phi(\bar{h}, g^c)$, where $\bar{h}$ represents the preferences of the "average consumer" and g^c is a specialization for consumer c. A key benefit to multi-task learning is that given $\bar{h}$, learning each g^c is easier than learning h^c directly. Thus, multi-task learning reduces the burden of learning on each consumer, by requiring fewer questions, and improving overall performance.

Multi-Objective Usually, however, things are not as simple as that. Often our goal is more than just to obtain correct predictions. If we send advertisements to consumers that are likely to prefer the products, we want to minimize the cost of sending ads to consumers that will never buy the product. If the cost of ineffective advertising is not known directly, we can analyze the tradeoffs to help us set a pricing scheme. Finally, we can create groups of consumers that compete for prediction performance.

Advertising Cost Often there are different costs associated with each prediction. Suppose we receive \$10 in profit when a consumer c buys a product x from some advertisement, but we lose \$1 if c does not buy x. Thus, there is tradeoff between two objectives: maximizing the rate of predicting L when the consumer prefers the product and minimizing the prediction rate when the product is not preferred. Often these costs are not known in advance, so we would like to see the tradeoffs. The tradeoff surface helps us decide whether or not to send the advertisement.

Consumer Groups Suppose that we have two groups of consumers, premium and regular. Premium consumers can pay for what we could call "more weight" in the learning process, meaning that it is more important to obtain good performance for premium consumers even at the expense of worse performance for regular consumers. This can be achieved by viewing each group as an objective. It may be the case that there is no solution that is best for both groups, so we will prefer a better solution for the premium group. By analyzing the tradeoff surface, we can establish a rate of exchange between the groups to determine a pricing scheme.

1.5 Multi-Objective Optimization

Our description of multi-objective optimization closely follows [29]. Theorems and claims from [29] will be stated without proof for brevity.

1.5.1 Framework

In multi-objective optimization, the notion of optimality is defined in terms of the optimization variables and the values of the objective functions. We denote the optimization variables by x and the objective values by y. We model a multi-objective optimization problem as:

$$\min_{x \in \mathbb{R}^n} \quad (f_1(x), \ldots, f_k(x))$$
$$x \in X \subseteq \mathbb{R}^n$$

where k is the number of objectives, $f_i : \mathbb{R}^n \to \mathbb{R}$ is an objective function, and $X \subseteq \mathbb{R}^n$ is known as the feasible set, which represents the set of all valid values of x. An unconstrained optimization problem occurs when $X = \mathbb{R}^n$, and a constrained problem occurs when $X \subset \mathbb{R}^n$. In this work, we will deal only with

constrained optimization problems. The value of each objective function for each feasible point $x \in X$ is known as the objective set Y:

$$Y \quad = \quad \{(f_1(x), \ldots, f_k(x)) \mid x \in X\}$$

The conflicting nature of multi-objective problems has important implications for multi-task learning. Basically, there is no perfect solution, so the best one can hope for is a compromise. A multi-objective problem can be divided into two classes, those for which objectives conflict and those for which the objectives are compatible. Let each $X_i^\star$ be the set of solutions to the following set of single-objective problems:

$$\begin{aligned} \min_x \quad & f_i(x) \\ & x \in X \end{aligned}$$

where $1 \leq i \leq k$. Note that for convex problems, the set $X_i^\star$ is a singleton. We say that the objectives are in conflict if and only if:

$$X_1^* \cap \cdots \cap X_k^* \quad = \quad \emptyset$$

meaning that there is no $x \in X$ that simultaneously optimizes all objectives. The objectives are compatible if there is some $x \in X$ that does satisfy all objectives.

1.5.2 Efficiency and Dominance

If the objectives are in conflict, then we are interested in finding and comparing compromises. Unfortunately, most problems have a continuum of compromises, which constitute the tradeoff surface. We compare solutions in terms of their objective values.

(1.1) **Definition**: Let $x_1, x_2 \in X \subseteq \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^k$, $f(x_1)$ *dominates* $f(x_2)$ if, for $1 \leq i \leq k$:

$$f_i(x_1) \quad \leq \quad f_i(x_2)$$

We also say that x_1 *dominates* x_2 if $f(x_1)$ dominates $f(x_2)$.

We write this relation as $x_1 \leq x_2$ and $f(x_1) \leq f(x_2)$. The set of all undominated points is denoted Y_N and is called the undominated set or the tradeoff surface.

(1.2) **Definition**: A feasible point $\hat{x} \in X$ is *efficient* or Pareto optimal if there is no other $x \in X$ such that $f(x) \leq f(\hat{x})$. The set of all efficient solutions is denoted by X_E and called the efficient set.

We can define the undominated set formally as:

$$Y_N \quad = \quad \{f(x) \mid x \in X_E\}$$

An equivalent definition of an efficient point is that $\hat{x}$ is efficient if there is no $x \in X$ such that $f_i(x) \leq f_i(\hat{x})$ for $i = 1, \ldots, k$ and $f_i(x) < f_i(\hat{x})$ for some $i \in \{1, \ldots, k\}$.

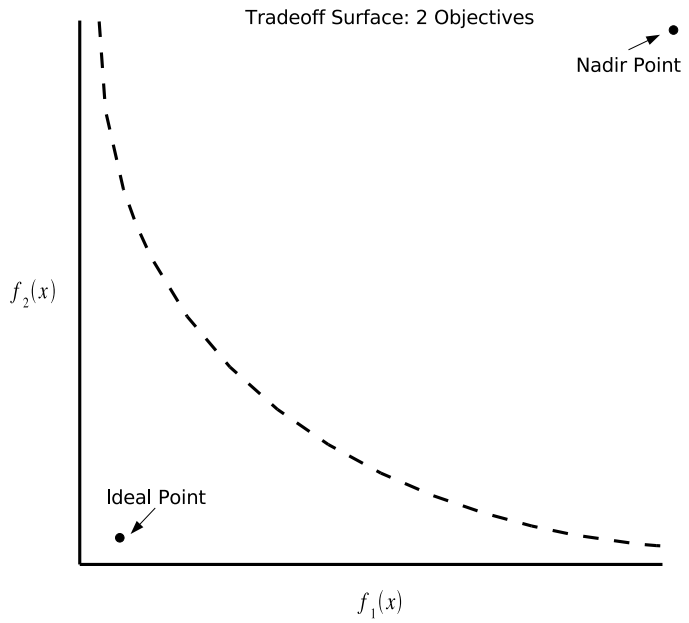


Figure 1.1: An example tradeoff surface, indicating the ideal point (lower-left) and the nadir point (upper-right). Since the objectives are in conflict, the ideal point dominates all other efficient points but is not contained in the tradeoff surface.

1.5.3 Tradeoff Surfaces

One goal of multi-objective optimization is to find the efficient set X_E, which constitutes the tradeoff surface. Rather than returning the single solution $x^\star$, an algorithm returns the set X_E. In order to intuitively understand the efficient set and the undominated set, we can show a tradeoff surface for two objectives, such as the one in Figure 1.1. This figure shows the convex hull of the augmented undominated set, $Y_N \cup y_N$, where y_N is called the nadir point. The nadir point, in two dimensions, represents the worst possible value for each objective function. The point in the lower-left corner of the plot is the ideal point y_I. This point corresponds to the set $X^\star$, the value of the optimal solution for each objective. Graphically, we see that if the objectives are not in conflict, then the ideal point is on the tradeoff surface. If the objectives are in conflict, as shown in the figure, then the ideal point dominates all other points in the undominated set. It is not on the tradeoff surface, meaning that there are no efficient points leading to y_I.

1.5.4 Scalarization Methods

Rather than solving for all efficient points, we are sometimes satisfied with just one efficient point. Typically, we can find this point by weighting the different objectives. This transforms the multi-objective problem into a single-objective problem which is easier to solve in some cases. This technique is called scalarization, though in the machine learning literature it is similar to regularization.

Objective Weights

A scalarized multi-objective problem requires a weight vector $\lambda \in \mathbb{R}^k$. The multi-objective optimization problem can then be written as:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \textstyle\sum_{i=1}^{k} \lambda_i f_i(x) \\ & x \in X \end{aligned}$$

When $\lambda_i > 0$, the solution $x^\star$ to this single-objective problem is an efficient point in the original problem.

(1.3) **Theorem** [29]: If the constraints X are a convex set and each objective f_i is convex, then for any weakly efficient point $\hat{x} \in X_{wE}$, there exists some non-negative weighting vector λ such that $\hat{x}$ is an optimal solution to the weighted multi-objective problem.

The (weakly) efficient point found from the scalarized problem with weight vector λ corresponds to the point on the tradeoff surface that is tangent to a hyperplane with weights λ. When the tradeoff surface is convex, λ defines a supporting hyperplane of the set.

Regularization

Regularization, in the context of multi-objective optimization, is a special case of scalarizing multiple objectives. In the standard formulation of the single-task support vector machine (SVM) classifier, the objective is:

$$\min_{w \in \mathbb{R}^q, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \quad \tfrac{1}{2} \|w\|^2 + \tfrac{C}{n} \textstyle\sum_{j=1}^{n} \xi_j$$

where w is the vector of weights for the separating hyperplane and ξ is a set of slack variables. This formulation is typically explained in the context of regularization theory [69]. We want, first of all, to minimize the slack variables ξ. For some datasets, however, there are many weight vectors w that achieve this minimum error, thus making the problem ill-posed. From this set of weight vectors, we would like to select the one with the smallest norm. In the context of multi-objective optimization, we see that this problem is simply a scalarized version of the following multi-objective problem:

$$\min_{w \in \mathbb{R}^q, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \quad \left(\|w\|^2 , \textstyle\sum_{j=1}^{n} \xi_j \right)$$

with objective weights $\left(\frac{1}{2}, \frac{C}{n}\right)$. If the data is guaranteed to be separable, then we are guaranteed that all slack variables ξ are zero. In this case, we are interested in the lexicographic minimization problem:

$$\min_{w \in \mathbb{R}^q, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \quad \left(\tfrac{1}{n} \textstyle\sum_{j=1}^{n} \xi_j, \|w\|^2 \right)$$

where the $\min$ operator is the lexicographic minimizer, stating that we want to minimize the error first and then minimize the norm of the weights. In the inseparable case, however, we do not make such strong requirements on the objective functions, seeking instead an efficient point.

The benefit of the multi-objective approach to a regularization problem would be to obtain the tradeoff surface and then find the objective weights. The tradeoff surface will give us the undominated set and efficient points. We can then select an appropriate regularization parameter C as a supporting hyperplane of the undominated set.

1.6 Related Learning Frameworks

We now formally define two related learning frameworks, which provide a foundation for our proposed multi-objective multi-task learning framework. Although the goal of each is to learn a function mapping examples to class labels, they differ on the structure of the input as well as the objectives to optimize. Single-task (single-objective) learners assume that the input is a sample from some distribution over examples, and the goal is to learn one function to minimize expected loss. Multi-task learners assume that the input consists of one sample from several distributions, and the goal is to learn multiple functions, one for each task, minimizing average expected loss.

1.6.1 Single Task

In single-task learning, the inputs consist of a vector of features. Each vector is called an example, which we denote as x. The input space is the set of all possible examples, denoted as $\mathcal{X}$, such that $x \in \mathcal{X}$. Often, we will assume that $\mathcal{X} = \mathbb{R}^q$ so that $x \in \mathbb{R}^q$. Associated with each example $x \in \mathcal{X}$ is a class label $y \in \mathcal{Y}$. In classification, the set of class labels $\mathcal{Y}$ is finite, and in regression $\mathcal{Y} \subseteq \mathbb{R}$. A dataset, $X \subseteq \mathcal{X}$, consists of n (x, y) pairs sampled identically and independently (i.i.d.) from a fixed but unknown probability distribution T on $\mathcal{X} \times \mathcal{Y}$, which is known as the learning task.

A learning algorithm outputs a function (called a hypothesis) $h : \mathcal{X} \rightarrow \mathcal{Y}$, mapping all possible examples to a class label. The algorithm is trained on a dataset X, containing n examples. We call the class of possible functions the hypothesis space $\mathcal{H}$, such that $h \in \mathcal{H}$. A hypothesis space is part of a family of hypothesis spaces, $\mathbb{H}$, such that $\mathcal{H} \in \mathbb{H}$.

(1.4) **Definition**: Let $\mathcal{X}$ be the input space, $\mathcal{Y}$ be the output space, and $\mathcal{H}$ be a hypothesis space. A *single-task learning algorithm* is any function A such that:

$$A : \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^n \rightarrow \mathcal{H}$$

The meaning of this definition is that the function A outputs a function $h \in \mathcal{H}$ given a training set of n examples.

This definition of a learning algorithm says nothing about the performance of that algorithm. By performance, we mean the ability of the learned hypothesis h to generalize to new examples sampled from the same distribution T, minimizing an expected loss functional, $L(h, T)$:

$$L(h, T) = \int_{\mathcal{X} \times Y} l(h(x), y) dT(x, y)$$

where $l : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ is called the loss function. Since the true probability distribution, T, is unknown, we cannot in general minimize the expected loss directly. Instead, we seek a hypothesis $h^{\star}$ that minimizes the empirical loss on the dataset X:

$$L(h, X) = \frac{1}{n} \sum_{j=1}^{n} l(h(x_j), y_j)$$

which is the average loss for all examples in the training set X. This strategy is known as the empirical loss (risk) minimization inductive principle (ERM). We must then determine whether following this strategy actually leads us to a solution that minimizes the expected loss in the limit of an infinite number of examples. If this is the case, then the strategy is said to be consistent [76].

(1.5) **Definition** [76]: Minimizing empirical loss is *consistent* for the set of functions $h \in \mathcal{H}$, loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, and the probability distribution function T on $\mathcal{X} \times \mathcal{Y}$ if the following two sequences converge in probability to the same limit:

$$\begin{aligned} L(h_n, T) &\longrightarrow \inf_{h \in \mathcal{H}} L(h, T) \\ L(h_n, X) &\longrightarrow \inf_{h \in \mathcal{H}} L(h, T) \end{aligned}$$

where the limit is as $n \to \infty$.

Here h_n denotes the hypothesis that minimizes the empirical loss for a dataset with n examples. Since this strategy has been shown to be consistent, most single-task learning algorithms follow it or some variation of it.

1.6.2 Multiple Tasks

In multi-task learning we want to find a hypothesis space $\mathcal{H}$ for a fixed set of tasks $T = \{T_t\}$. We are given a set of m datasets, X^t for $1 \leq t \leq m$, each containing n samples. Each dataset X^t is sampled from a distribution T_t, each of which is sampled from a hyper-distribution, Φ. We want to learn m hypotheses, one for each task. We assume that these hypotheses are drawn from the same space $\mathcal{H}$ belonging to a family of hypothesis spaces, $\mathbb{H}$.

(1.6) **Definition**: A *multi-task learning algorithm* learns a hypothesis space $\mathcal{H} \in \mathbb{H}$ from a sample of m datasets, each with n examples:

$$A : \bigcup_{m=1}^{\infty} \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^{mn} \to \mathbb{H}$$

where each $X \in (\mathcal{X} \times \mathcal{Y})^{mn}$ refers to m datasets each consisting of n examples.

Regardless of the method for obtaining a single hypothesis for each task, we associate each hypothesis space θ with the vector of hypotheses $(h_1, \ldots, h_m)$. That is, given a shared hypothesis space $\mathcal{H}$, there is some single-task learning algorithm that can find a single hypothesis, h_t for each task, leading to a vector of hypotheses $(h_1, \ldots, h_m)$, $h_t \in \mathcal{H}$. The expected and empirical loss for the hypothesis space depends on this vector.

$$\begin{aligned} L(\mathcal{H}, \{T\}) &= \frac{1}{m} \sum_{t=1}^{m} L(h_t, T_t) \\ L(\mathcal{H}, X) &= \frac{1}{m} \sum_{t=1}^{m} L(h_t, X_t) \end{aligned}$$

where we use summation for the expected loss rather than an integral because we assume that the sequence of tasks is fixed. That is, we are interested in fitting these tasks only and not all tasks from the hyper-distribution Φ. This model of multi-task learning differs from that of Baxter and more closely resembles the work of Maurer [13, 58].

Multi-task algorithms can be divided into two groups. In common-model algorithms, a single-task learning algorithm is divided into two parts, one shared between tasks and one specialized for each task. In common-transformation algorithms, each task is transformed by a common operator, which is learned.

Common Model

The common model approach learns a hypothesis h for each dataset X^t individually, but it is divided into parts:

$$h_t \quad = \quad \phi(\bar{h}, \hat{h}_t)$$

where $\hat{h}_t$ is a specific to each dataset and $\bar{h}$ is a common component of the hypothesis, shared across all datasets. The function ϕ indicates a computationally simple method for creating a composite model from the two components.

In the common model approach, once a decomposable algorithm is found, one must balance the tendency to make all hypotheses be the same with the tendency to specialize. Ideally, of course, all tasks can be learned equally well with the same model, meaning that the component $\hat{h}_t$ contribute little to the overall solution. Unfortunately, this is rarely the case. If all tasks are forced to have the same model (a high degree of commonality), the generalization performance may be poor due to over training across all tasks. Conversely, if we assume that there is no commonality, then the individual tasks may not contain enough examples for learning, resulting in poor generalization performance.

Common Transformation

The transformation approach finds a transformation operator F such that the transformed task, $T'_t = F(T_t)$, is easier to learn. For example, if F is a vector of binary values, then applying F to a task is equivalent to selecting a subset of features. The goal of transformation-based algorithms is to learn F given a set of tasks, $T = \{T_t\}$. A potential benefit of these methods is that existing algorithms can be used once a transformation is available.

In the transformation approach, the main tradeoffs are more difficult to quantify. Ideally, we can find a projection in which each task is mapped to a perfectly separable task and learning is trivial. This can be difficult to achieve because a good transformation for one task may not be suitable for another. As a result, the tradeoff is between the complexity, which we will define later, and the loss of learning the transformed task.

1.7 Multi-Objective Multi-Task Learning

In contrast to single-objective multi-task learning, multi-objective multi-task learning algorithms seek a set of efficient hypothesis spaces, Pareto-optimal with respect to a vector-valued loss function.

(1.7) **Definition**: A *multi-objective multi-task learning algorithm* learns a set of hypothesis spaces Θ, where $\Theta \subseteq \mathbb{H}$:

$$A: \ \bigcup_{m=1}^{\infty} \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^{mn} \rightarrow \ \ 2^{\mathbb{H}}$$

where $2^{\mathbb{H}}$ denotes the set of all subsets of hypothesis spaces.

We denote the set of hypothesis spaces returned by the algorithm as $\Theta \subseteq \mathbb{H}$. The empirical tradeoff surface is defined as:

$$\hat{L} \quad = \quad \{L(\mathcal{H}, X) \mid \mathcal{H} \in \Theta\}$$

which is the vector-valued loss of each hypothesis space. The expected tradeoff surface is defined as:

$$L \quad = \quad \{L(\mathcal{H}, T) \mid \mathcal{H} \in \Theta\}$$

where the function L is the same as in single-objective multi-task learning but defined for a vector-valued loss function. From the definition of the algorithm, we cannot infer if the hypothesis spaces are efficient.

1.7.1 Empirically Efficient Learning

The empirically efficient learning principle requires that an algorithm minimize all its objectives on the training set. An empirically efficient multi-objective multi-task learning algorithm outputs at least one efficient hypothesis space given a finite set of datasets. In single-task learning, the analogous principle is empirical loss minimization. Any single-task learning algorithm that minimizes the loss on the training set was shown to converge to the optimal expected loss as the sample size increases [76]. Clearly, this extends to multiple loss functions so that any multi-task learning algorithm that finds an empirically efficient solution will find an efficient solution as the sample size increases.

The algorithm learns the empirical efficient set and is expected to converge to the expected empirical set as the sample size increases.

(1.8) **Definition**: Let $\theta \in \mathbb{H}$ be a hypothesis space, X be a multi-task dataset, and L be the multi-objective multi-task loss function. The *empirical efficient set* $\hat{\Theta}_E$ is defined as:

$$\hat{\Theta}_E \quad = \quad \left\{\theta \mid L(\mathcal{H}, X) \leq L(\mathcal{H}', X)\, \mathcal{H}' \in \mathbb{H}\right\}$$

and the *empirical tradeoff surface* $\hat{L}_N$ is defined as:

$$\hat{L}_N \quad = \quad \left\{L(\mathcal{H}, X) \mid \mathcal{H} \in \hat{\theta}_E\right\}$$

The empirical tradeoff surface is just the undominated set with respect to the empirical efficient set.

(1.9) **Definition**: Let $\mathcal{H} \in \mathbb{H}$ be a hypothesis space, $T = \{T_t\}$ be a sequence of tasks, and L be the multi-objective multi-task loss function. The *expected efficient set* Θ_E is defined as:

$$\Theta_E \quad = \quad \left\{\mathcal{H} \mid L(\mathcal{H}, \{T\}) \leq L(\mathcal{H}', T)\, \mathcal{H}' \in \mathbb{H}\right\}$$

and the *expected tradeoff surface* L_N is defined as:

$$L_N \quad = \quad \{L(\mathcal{H}, T) \mid \mathcal{H} \in \Theta_E\}$$

These definitions directly extend the empirical and expected multi-task loss to multiple objectives. Since several objectives may not share a common solution, there is a set of equally good solutions. The set can have many, possibly a continuum of, points.

We define an empirically efficient multi-objective multi-task learning algorithm as one that returns a set of hypothesis spaces contained in the empirical efficient set given a fixed sequence of datasets.

(1.10) **Definition**: For some finite multi-task sample $X \in (\mathcal{X} \times \mathcal{Y})^{mn}$, an *empirically efficient multi-objective multi-task learning algorithm* is defined as:

$$A : \bigcup_{m=1}^{\infty} \bigcup_{n=1}^{\infty} (\mathcal{X} \times \mathcal{Y})^{mn} \quad \rightarrow \quad 2^{\mathbb{H}}$$

such that

$$A(X) \quad \subseteq \quad \hat{\Theta}_E$$

In the case that there is only one objective, this principle reduces to the empirical loss minimization principle. The definition only requires a subset of efficient solutions, so any algorithm that returns just one efficient hypothesis space would suffice. Hence, all algorithms that optimize a weighted combination of convex objectives, such as regularization algorithms, are empirically efficient. Ideally, however, we would prefer to find all efficient hypothesis spaces. As we will see in Chapter 3, many existing multi-objective multi-task learning algorithms that are empirically efficient do not return all efficient points.

The empirically efficient learning principle generalizes the single-objective learning principle. To illustrate this, let k be the number of objectives and m be the number of tasks. When $k = 1$ and $m > 1$, we have single-objective multi-task learning. When $k > 1$ and $m = 1$, we have single-task multi-objective learning. Finally, when $k = 1$ and $m = 1$, we have the standard single-objective, single-task learning.

1.7.2 Weighted Objectives

Creating a scalar objective function from several objectives using a vector of weights is common in machine learning, but it is not typically recognized as multi-objective optimization. In particular, this is applied to guard against over-fitting the dataset, which is a frequent problem. By applying a high-capacity learner, we can fit a training set well, and in many cases perfectly well. When testing the learned hypothesis, however, we often find that the loss is higher than was expected. This phenomenon is known as over-fitting. A common solution is to instead control the capacity of the learned hypothesis, to regularize it, to reach a compromise between a low-capacity hypothesis and an accurate hypothesis. For example, let $\|\mathcal{H}\|$ denote the capacity of the hypothesis space and let the loss function L be scalar-valued, then the optimization problem could be written as:

$$\min_{\mathcal{H}} \quad (L(\mathcal{H}, X), \|\mathcal{H}\|)$$

meaning that we want to minimize the empirical loss and the capacity of the hypothesis space.

The empirical tradeoff surface for this problem helps us find a weight for the regularization term, $\|\mathcal{H}\|$. The expected tradeoff surface is of little use to us, however, because the regularization term is constant. It is also unique to the hypothesis family, so comparing the tradeoff surfaces of two different algorithms is not possible.

Since we are interested in an objective weight, we really want to solve a single-objective problem by scalarizing the objectives. A composite loss function is defined as follows:

(1.11) **Definition**: Let $\lambda \in \mathbb{R}^k$ with $\lambda_i \geq 0$ and let $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^k$ be such that $l = (l^{(1)}, \ldots, l^{(k)})$. A *composite loss function* $l_\lambda : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^k$ is such that:

$$\begin{aligned} l_\lambda(y, y') &= \lambda^T (l^{(1)}(y, y'), \ldots, l^{(k)}(y, y')) \\ &= \sum_{i=1}^{k} \lambda_i l^{(i)}(y, y') \end{aligned}$$

for $y, y' \in \mathcal{Y}$.

With this composite loss function, we can treat the multi-objective problem as a single-objective problem.

1.7.3 Multiple Loss Functions

Another common use of multi-objective optimization in machine learning is to minimize multiple data-dependent loss functions. In binary classification, there are two such objectives, minimize the error in each class. The standard solution to this problem is to combine these functions into a single objective: maximize

the area under the ROC curve [1]. Although this was shown, in the single-task case, to lead to optimal ROC curves, it is not a multi-objective approach. In the multi-objective approach, we assume that the loss functions are averaged over all datasets with the following form, for $k \geq 1$:

$$L(\mathcal{H}, X) \quad = \quad (L_1(\mathcal{H}, X), \ldots, L_k(\mathcal{H}, X))$$

Where each component loss function, L_i, is a scalar-valued function. The optimization problem then has the following form:

$$\min_{\mathcal{H}} \quad L(\mathcal{H}, X)$$

The empirical and expected tradeoff surfaces are directly comparable as are the tradeoff surfaces between different algorithms. For example, we can optimize the same objectives as plotted in a ROC curve. The resulting tradeoff surface will resemble the ROC curve but with one key difference. With ROC curves, a threshold is selected for a single hypothesis. In our formulation, however, each point in the tradeoff surface is a different hypothesis space.

1.7.4 Task Clusters

When there is no single hypothesis space that is optimal for all learning tasks, we can balance the loss values in clusters of tasks. This is different from our previous frameworks. Earlier, we sought a hypothesis space that balances several different loss function, averaged across all tasks. An implicit assumption in these and most other frameworks is that the optimal hypothesis space contains the optimal hypothesis for each task. If this assumption fails to hold, then there will be a tradeoff between fitting one task versus another.

The loss function used now considers each task, or cluster of tasks, as a separate objective:

$$L(\mathcal{H}, X) \quad = \quad \big(L(\mathcal{H}, X^1), \ldots, L(\mathcal{H}, X^m)\big)$$

Now the loss function has dimensionality equal to the number of tasks. We can analyze this framework as a single-objective loss function, which is a scalarization of this multi-objective loss function with weight vector $\lambda_i = \frac{1}{m}$. An example of this framework would be to minimize the error rate in two clusters of tasks. We assume that we can partition the tasks into 2 clusters and that g_i contains the indices of the tasks in cluster i. The objectives for this problem are:

$$\min_{\mathcal{H}} \quad \Big(\tfrac{1}{|g_1|}\textstyle\sum_{t \in g_1} L(\mathcal{H}, X^t), \tfrac{1}{|g_2|}\textstyle\sum_{t \in g_2} L(\mathcal{H}, X^t)\Big)$$

The tradeoff surface for this problem tells us if the tasks are symmetric. Symmetry in this context refers to whether we can improve the loss of cluster g_1 by δ if we degrade the loss of cluster g_2 by δ. If the tasks are asymmetric, improving one cluster could require a significant sacrifice from the other.

2 Theory

In our proposed learning principle, empirically efficient learning, the solution we seek is a set of efficient hypothesis spaces $\Theta_E = \{\mathcal{H}\}$. This is in contrast to single-objective multi-task learning, in which we seek a single optimal hypothesis space $\mathcal{H}$ that minimizes the expected loss across all tasks. In this section, we establish the terminology and show that the proposed learning frameworks have solid theoretical foundations.

Figure 2.1 shows the dependencies between the theoretical results presented in this chapter. Several new theorems are proposed that establish that the empirically efficient learning is consistent. These bound the sample complexity for the weighted objectives and multiple loss function frameworks. The data-dependent bounds depend on the Rademacher complexity formulation for multi-task learning developed by Maurer [58]. The new bounds are derived by bounding the distance between tradeoff surfaces by the distance between one point on each surface. Then, the Rademacher complexity of each new loss function is bounded based on the range of the function, by means of a Lipschitz constant. The two-sided, data-independent, bounds that we derive depend only on the covering over the set of loss values, which is in this case a vector space. Given a covering whose size depends on the sample size, we next bound the probability of a large error using a newly derived multi-variate form of Chebyshev's inequality. In all cases, the theorems reduce to the existing versions when the number of objectives is set to $k = 1$. However, the case when $k > 1$ is of interest in the present work.

2.1 Sample Complexity Results

Although we have defined our learning framework in terms of the empirically efficient learning principle, we would like to know how many tasks and examples are necessary to converge to an expected efficient point. We study the sample complexity of three types of multi-objective multi-task learning algorithms. The complexity results show that the number of examples of each task necessary to achieve a given error level increases, in most cases, sub-linearly with the number of objectives.

In analyzing the sample complexity of an algorithm, we are interested in the number of examples that must, at worst, be provided to the learner in order to achieve some error value. Most existing results in the literature are the so-called data-independent bounds, such as those involving the VC dimension [76]. These concern only the capacity of the hypothesis space, where the capacity measure is intuitively similar to the number of hypotheses in the space. As the number of possible solutions increases, we tend to need more examples to narrow the space down to a single answer. The sample complexity tells us, asymptotically, how the number of examples needed depends on the capacity of the hypothesis space. In multi-task learning, we are concerned not only with the number of examples per task but also the number of tasks. Here, we will also be interested in how the number of objectives affects the number of examples needed. The key questions we will consider are whether adding multiple objectives increases the sample complexity and if so, by how much.

Several theoretical frameworks have been recently proposed for multi-task learning, and we extend these results to handle multiple objectives. The main purpose of these frameworks is to show that multiple tasks reduce the overall sample complexity of the learner for each task [13, 14]. With the advent of data-dependent bounds, we can obtain tighter, though one-sided, bounds on the sample complexity [2, 57, 58]. Our goal for this section is not to obtain extremely tight bounds but to show that adding additional objectives does not

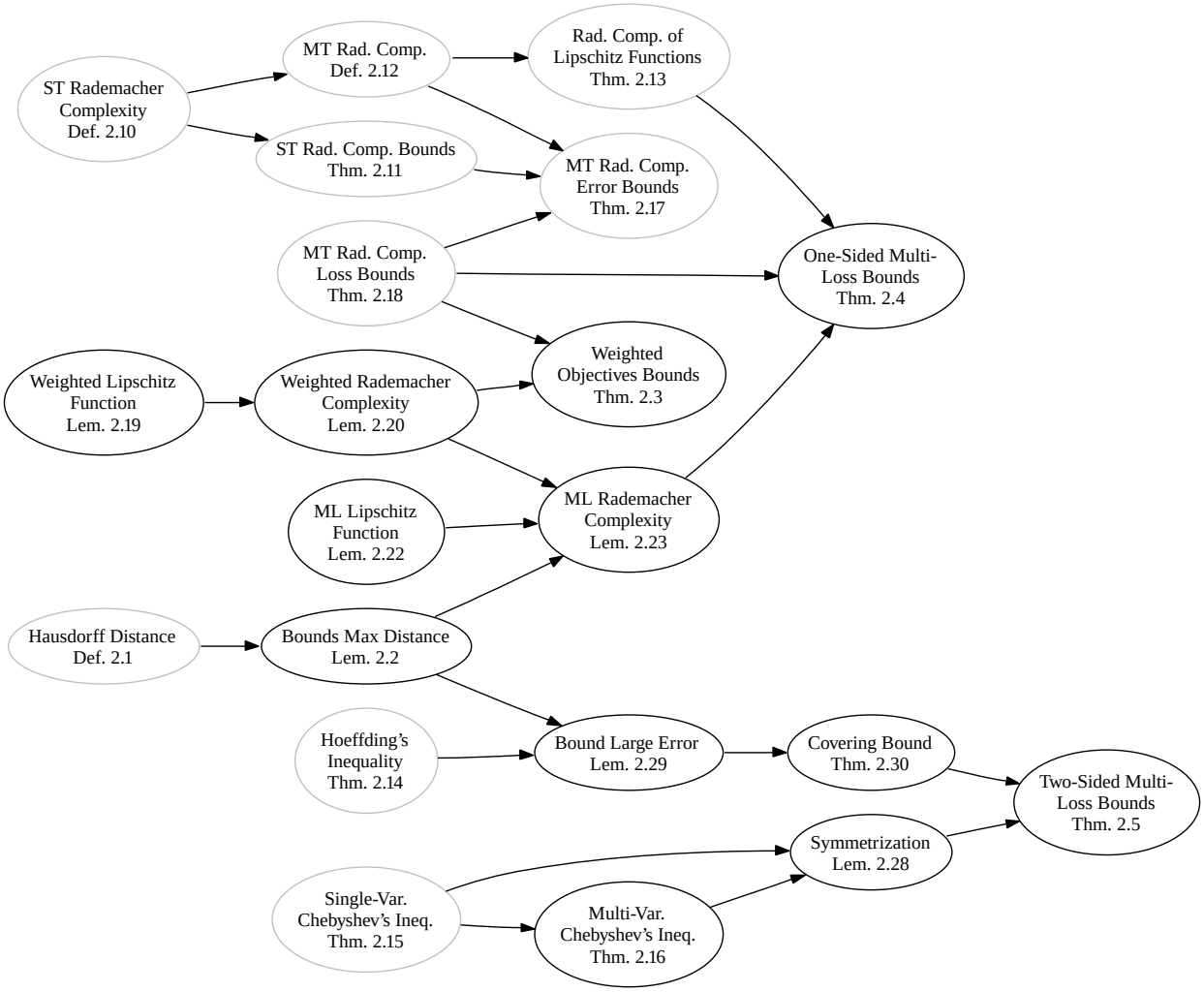


Figure 2.1: Dependencies among the theoretical results. New results are in black. Results reproduced from existing work are in gray.

dramatically increase the sample complexity.

2.1.1 Types of Bounds

Sample complexity results bound the difference between the empirical loss and the expected loss for some class of learning algorithms. Note that the bounds do not in any way concern the optimal loss, known as the Bayes error rate [28]. That is, although a bound guarantees convergence to the expected loss, it does not guarantee that the expected loss is anywhere near the optimal loss.

Adding multiple objectives complicates the analysis of sample complexity in two ways. First, we now have vector-valued loss functions, so some of the foundational results, such as the Chebyshev and Hoeffding inequalities, have to be examined more closely. Second, we have a set of solutions, the efficient, so we want to bound the distance between two sets of tuples rather than the difference between two scalars.

Convergence is defined in terms of the distance between the empirical and expected undominated sets. We want to bound the worst possible distance by some constant ϵ that depends on the number of examples, the size of the hypothesis family, etc. The bound should have the following form:

$$d(L_N, \hat{L}_N) \quad \leq \quad \epsilon(m, n, \mathbb{H}, \mathcal{L})$$

where ϵ is some function, $\mathcal{L}$ is the class of loss functions, $\mathbb{H}$ is the hypothesis family, m is the number of tasks, and n is the number of examples in each task. Obviously, this function should be decreasing in m and n. It is usually expected to increase in the capacity of the hypothesis family, $|\mathbb{H}|$, with respect to some capacity measure. A natural choice for the distance metric d is the Hausdorff distance [21]. The Hausdorff distance is a semi-metric over the set of compact subsets of $\mathbb{R}^k$ where k is the number of objectives.

(2.1) **Definition** [21]: Let $A, B \subseteq \mathbb{R}^k$ be compact sets and $r \in \mathbb{R}$, $r > 0$. The *Hausdorff distance* is:

$$d_H(A, B) \quad = \quad \max\{\delta^*(A, B), \delta_*(A, B)\}$$

where

$$\begin{aligned} \delta^*(A, B) &= \inf\{r > 0 : B \subset N_r(A)\} \\ \delta_*(A, B) &= \inf\{r > 0 : A \subset N_r(B)\} \end{aligned}$$

and $N_r(A)$ is the neighborhood of radius r around the set A, assuming some lower-level distance function $d : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}_{+,0}$:

$$N_r(A) \quad = \quad \left\{ x \in \mathbb{R}^k \mid d(x, a) \leq r, a \in A \right\}$$

The function δ^* returns the smallest radius that can be added around each point in A such that the set B is a proper subset of the enlarged set A; δ_* is defined similarly. The advantage of the Hausdorff distance for our analysis is that the distance between tradeoff surfaces can be reduced to a distance between some pair of points. As a result, we can restrict our analysis to pairs of points, drawing on existing results in the literature. The following lemma relates bounds on each point in the set to the Hausdorff distance.

(2.2) **Lemma**: Given $r > 0$, if for each $a \in A$ $\exists b \in B$ such that $d(a, b) \leq r$ and for each $b \in B$ $\exists a \in A$ such that if $d(b, a) \leq r$, then $d_H(A, B) \leq r$.

Proof: The first condition gives us that $\delta_*(A, B) \leq r$, since for every $a \in A$ there is some $b \in B$ such that $a \in N_r(b)$; thus, $A \subset N_r(B)$. The second condition gives us $\delta^*(A, B) \leq r$ since for each $b \in B$, there is some $a \in A$ such that $b \in N_r(a)$, so $B \subset N_r(A)$. Thus, $\max\{\delta^*, \delta_*\} \leq r$, so $d_H(A, B) \leq r$.

□

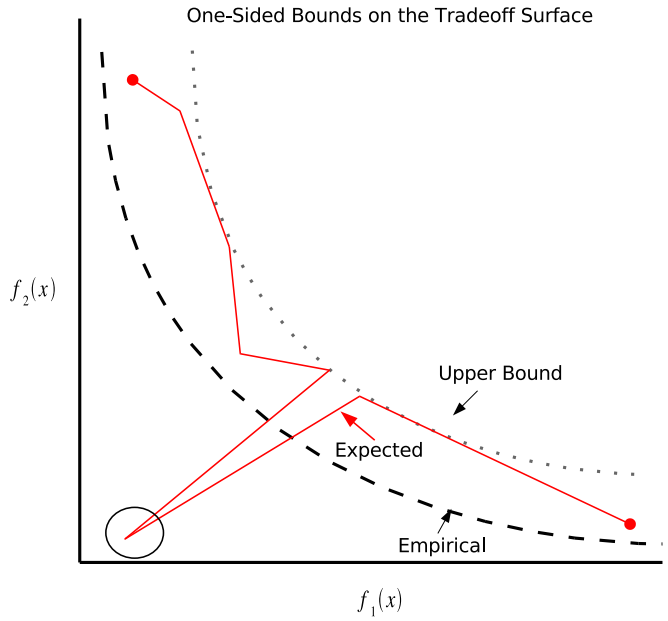


Figure 2.2: One-sided bounds are only effective if we know that the expected tradeoff surface does not dominate the empirical tradeoff surface. They establish only an inner surface of maximum deviation.

One-Sided

Most existing bounds for multi-task learning are called one-sided bounds. They are defined for each $\mathcal{H} \in \mathbb{H}$, such that the empirical loss (plus a constant) bounds the expected loss from above:

$$L(\mathcal{H}, T) \quad \leq \quad L(\mathcal{H}, X) + \epsilon(m, \mathbb{H}, \mathcal{L})$$

where we assume that there is only 1 objective, $k = 1$. They are called one-sided because, in this case, there is no lower bound so $L(\mathcal{H}, T)$ could be arbitrarily small. One-sided bounds are insufficient to bound the Hausdorff distance, as shown in Figure 2.2. If a point $\hat{l}$ in the empirical tradeoff surface dominates l in the expected tradeoff surface, we can bound the distance with one-sided bounds; however, if the l dominates $\hat{l}$, it could be arbitrarily close to the ideal point, having a large distance. In general, however, we do not know whether the empirical tradeoff surface dominates the expected tradeoff surface.

One-sided bounds do establish an upper-limit on the Hausdorff distance between subsets of the tradeoff surface, under some conditions. If any point in the expected tradeoff surface does not Pareto dominate the empirical tradeoff surface, then the bounds tell us how much deviation is allowed. If the tradeoff surface is convex, the one-sided bounds give us an inner surface, such that the empirical tradeoff surface dominates the inner surface. We are guaranteed that no point in the expected tradeoff surface is dominated by the inner surface; thus, the expected tradeoff surface lies between the inner surface and the origin.

Two-Sided

In order to totally bound the Hausdorff distance between the expected and empirical tradeoff surfaces, two-sided bounds are necessary. The existing two-sided bounds for multi-task learning must be extended for an underlying vector-valued loss function.

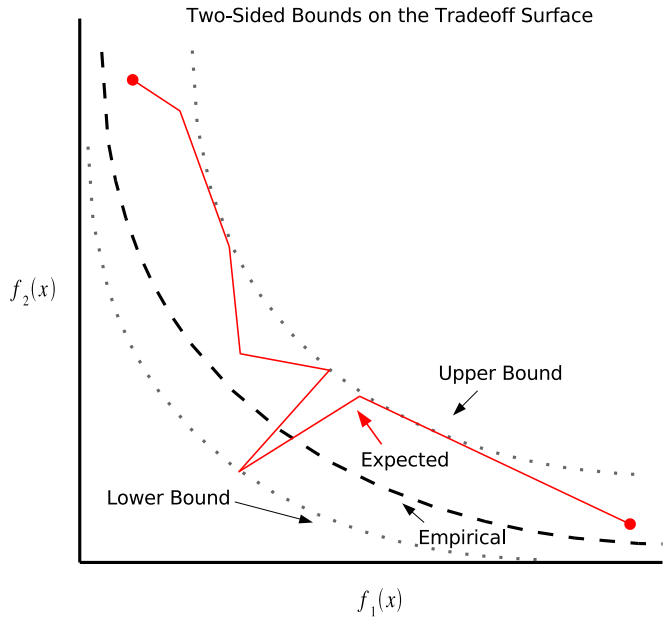


Figure 2.3: Two-sided bounds establish both an inner and outer surface of maximum deviation regardless of whether the expected tradeoff surface dominates the empirical tradeoff surface.

Since the Hausdorff distance is defined in terms of neighborhoods around points, a bound on the maximum distance around any point on the empirical tradeoff surface is a bound on the Hausdorff distance, which follows from Lemma 2.2 and is illustrated in Figure 2.3.

2.1.2 Weighted Objectives

We present one-sided data-dependent bounds for the most common form of multi-objective learning, weighted objectives. The composite loss function, from Definition 1.11, combines component loss functions with a weight vector λ. We prove the following theorem by extending Maurer's data-dependent multi-task bounds [58].

(2.3) **Theorem**: Let (X, T) be a (m, n) multi-task environment. For $\delta > 0$, $\gamma \in \mathbb{R}^k$ with $\gamma_i > 0$, $\lambda \in \mathbb{R}^k$ and $\lambda_i \geq 0$ for $1 \leq i \leq k$:

$$\frac{1}{m}\sum_{t=1}^{m} L(\mathcal{H}, T_t) \quad \leq \quad \frac{1}{mn}\sum_{t=1}^{m} L(\mathcal{H}, X_t) + \|\lambda * \gamma\| \, R_n^m(\mathbb{H}) + \sqrt{\frac{\ln\left(\frac{1}{\delta}\right)}{2mn}}$$

Proof: See Section 2.2.2.

These bounds depend on the magnitude of the weights λ and the range of each loss function, γ. As the number of objectives increases, the norm of the weight vector increases at a rate proportional to the square root of the number of objectives, assuming that the weights are equal. In addition, as with the original multi-task learning framework, the range on the loss function greatly influences the bounds.

2.1.3 Multiple Loss Functions

The most direct instantiation of the empirically efficient learning principle is the Pareto-optimal multi-task learner, which minimizes multiple loss functions. Here we bound the convergence to the vector-valued loss function for every hypothesis space. We prove the following one-sided bounds by extending Maurer's data-dependent framework [58].

(2.4) **Theorem**: Let (X, T) be a (m, n) be a multi-task environment. For $\delta > 0$, and $\gamma \in \mathbb{R}^k$ with $\gamma_i > 0$ for $1 \leq i \leq k$.

$$\frac{1}{m}\sum_{t=1}^{m} L(\mathcal{H}, T_t) \quad \leq \quad \frac{1}{mn}\sum_{t=1}^{m} L(\mathcal{H}, X_t) + \|\gamma\| \, R_n^m(\mathbb{H}) + \sqrt{\frac{\ln\left(\frac{1}{\delta}\right)}{2mn}}$$

Proof: See Section 2.2.3.

This theorem shows that the sample complexity increases proportionally to the Lipschitz constant of the vector-valued loss function, which increases linearly with the number of objectives. We prove the following two-sided bounds on the expected loss by extending Baxter's framework for bias learning based on a covering over the space of loss functions [13]. Here the loss function is assumed to be restricted in ranges to $[0, 1]^k$.

(2.5) **Theorem**[1]: Let $\mathcal{H} \subseteq \mathcal{H}_1 \oplus \cdots \oplus \mathcal{H}_n$ be a permissible class of functions mapping $(\mathcal{X} \times \mathcal{Y})^n \to [0, 1]^k$. Let z be an (n, m)-sample generated by sampling m times from $\mathcal{X} \times \mathcal{Y}$ according to T_t. For all $\nu > 0$ and $0 < \alpha < 1$. If

$$m \quad \geq \quad \max\left\{\frac{8\sqrt{k}}{\alpha^2 \nu n} \log \frac{4C\left(\frac{\alpha\nu}{8}, \mathcal{H}\right)}{\delta}, \frac{2k}{\alpha^2 \nu}\right\}$$

then

$$\Pr\left\{\mathbf{z} \in (\mathcal{X} \times \mathcal{Y})^{mn} : \sup_{\mathcal{H}} d_\nu\left[\hat{\mathrm{er}}_{\mathbf{z}}(\mathcal{H}), \mathrm{er}_{T_t}(\mathcal{H})\right] > \alpha\right\} \quad \leq \quad \delta$$

Proof: See Section 2.2.3.

These bounds clearly show that the number of examples increases at worst linearly with the number of objectives. The covering number here is defined over the vector-valued loss function, which is why it does not depend on the number of objectives explicitly.

2.2 Proof of Theorems

This section contains the proofs of the sample complexity theorems. It also contains some of the necessary background material that is difficult to find in the references.

2.2.1 Background

The theorems presented here extend existing results from other work in single-objective multi-task learning. Aside from background in single-task learning theory [3, 76] we include some background information necessary to understand the proofs.

[1] Extends Corollary 19 in [13].

Multi-Task Environment

We define a multi-task learning environment such that X is a multi-task dataset sampled from the set of tasks $T = \{T_t\}$. This definition will be used throughout this chapter.

(2.6) **Definition**: The pair (X, T) is an (m, n)-*multi-task environment* if $X \in (\mathcal{X} \times \mathcal{Y})^{mn}$, each X_t is sampled from a distribution T_t over $(\mathcal{X} \times \mathcal{Y})$, and each $T_t \in T$ is drawn from a hyper-distribution Φ.

The notation $(\mathcal{X} \times \mathcal{Y})^{mn}$ denotes all sets consisting of m datasets, each having n samples.

Covering Numbers

Covering numbers are used frequently in learning theory. Although the VC dimension is more popular in classification, covering numbers are seen as extensions to growth functions, which are used to derive the VC dimension [3, 76]. When the distance metric is clear from the context, it will be omitted.

(2.7) **Definition** (in [3]): For a set $X \subseteq \mathbb{R}^k$, constant $\epsilon > 0$, and distance metric d defined on X, $W \subseteq \mathbb{R}^k$ is an ϵ-*cover* for X if $W \subseteq X$ and for each $x \in X$, there exists a $w \in W$ such that:

$$d(w, x) \quad < \quad \epsilon$$

A covering of a set is a subset that is within some distance ϵ to each point in the set.

(2.8) **Definition** (in [3]): The number $C(\epsilon, X, d)$ is the *covering number* for a set X, constant $\epsilon > 0$, and distance metric d if:

$$C(\epsilon, X, d) \quad = \quad \min |W|$$

where W is an ϵ-cover of X.

The ϵ-covering number is the size of the smallest ϵ-cover, w, of the set X.

Lipschitz Functions

A frequently used result from real analysis is the idea of the Lipschitz function.

(2.9) **Definition** (in [21]): A *Lipschitz function* is any function $f : \mathbb{R}^n \to \mathbb{R}^k$ such that for some $\gamma \in \mathbb{R} > 0$ and for all $x, y \in \mathbb{R}^n$:

$$\|f(x) - f(y)\| \quad \leq \quad \gamma \|x - y\|$$

where the constant γ is called the Lipschitz constant.

Rademacher Complexity

Until recently, most convergence results for learning algorithms were data-independent, meaning that they depend only on properties of the algorithm, such as the VC dimension or covering numbers. It was later shown that one can derive data-dependent bounds as well. The benefit is that we can obtain tighter bounds given knowledge of the dataset. Unfortunately, it can be difficult to compute data-dependent bounds given a finite sample. The Rademacher complexity is both easy to compute and theoretically interesting for sample complexity. Its purpose is to bound the difference between the output of a learning algorithm trained on some dataset and the output of the same algorithm when trained on the same dataset whose class labels have been randomly flipped.

(2.10) **Definition** [18]: The *Rademacher average* (also called the Rademacher complexity) is defined for a class of functions $\mathcal{F}$ as:

$$\begin{aligned} \mathcal{R}(\mathcal{F}) &= \mathrm{E}\left[\sup_{f\in\mathcal{F}} R_n f\right] \\ R_n f &= \frac{1}{n}\sum_{i=1}^{n}\sigma_i f(Z_i) \end{aligned}$$

where n is the number of examples, $f \in \mathcal{F}$ is a function, Z_i is an example from the dataset, and σ_i is a Rademacher random variable. The random variable σ_i is independently drawn from $\{-1, 1\}$ with probability $\frac{1}{2}$ of taking each value. The conditional Rademacher average is defined as:

$$\mathcal{R}_n(\mathcal{F}) = \mathrm{E}_\sigma\left[\sup_{f\in\mathcal{F}} R_n f\right]$$

where the expectation is with respect to the Rademacher variable σ.

Here $\mathcal{R}(\mathcal{F})$ transforms the class of functions $\mathcal{F}$ into a different class of functions, still defined on the dataset X but computing the Rademacher complexity value instead of the expected value of $f \in \mathcal{F}$. The following theorem is the fundamental result regarding Rademacher averages, bounding the expected value of a function by its empirical estimate and a factor related to the Rademacher complexity.

(2.11) **Theorem** [18]: For all $\delta > 0$, with probability at least $1 - \delta$,

$$\begin{aligned} \forall f \in \mathcal{F},\, Pf &\leq P_n f + 2\mathcal{R}(\mathcal{F}) + \sqrt{\frac{\log\frac{1}{\delta}}{2n}} \\ \forall f \in \mathcal{F},\, Pf &\leq P_n f + 2\mathcal{R}_n(\mathcal{F}) + \sqrt{\frac{2\log\frac{2}{\delta}}{n}} \end{aligned}$$

The term Rademacher complexity usually refers to the quantity $2\mathcal{R}(\mathcal{F})$. For a single task, computing the Rademacher complexity is about as hard as computing the empirical loss. It can be computed as:

$$\mathcal{R}_n(\mathcal{F}) = 1 - 2\mathrm{E}\left[\inf_{f\in\mathcal{F}}\frac{1}{n}\sum_{i=1}^{n}\frac{1-\sigma_i f(X_i)}{2}\right]$$

which can be interpreted as running the learning algorithm on the dataset X but randomly flipping the class labels with probability $\frac{1}{2}$.

We extend the Rademacher complexity to multiple tasks by assuming that we have a vector of functions $f_t \in \theta$.

(2.12) **Definition** [58]: Let $(X, \{T\})$ be an (m, n) multi-task environment and let $(f_1, \ldots, f_m)$ be such that each $f_t \in \mathcal{F}$ and let σ_j^t be the Rademacher variable. The *multi-task Rademacher complexity* is:

$$\hat{R}_n^m(\mathcal{F})(x) = \mathrm{E}_\sigma\left[\sup_{f\in F}\frac{2}{mn}\sum_{t=1}^{m}\sum_{j=1}^{n}\sigma_j^t f_t(X_j^t)\right]$$

The following relation between Rademacher averages and Lipschitz functions is the foundation for several of our results [11].

(2.13) **Theorem**: Let F be an $\mathbb{R}^m$-valued function class on space X and suppose that $\phi : \mathbb{R} \to \mathbb{R}$ has a Lipschitz constant γ such that

$$\phi \circ F \quad = \quad \left\{ \left(\phi \circ f^1, \ldots, \phi \circ f^m \right) \mid \left(f^1, \ldots, f^m \right) \in F \right\}$$

then

$$\hat{R}_n^m(\phi \circ F) \quad \leq \quad \gamma \hat{R}_n^m(F)$$

As we will see later, Theorem 2.13 is useful in linking the complexity of the loss function defined on the output of the hypothesis to the output of the hypothesis itself. This requires that the loss functions obey a Lipschitz condition.

Inequalities

Several inequalities from the theory of empirical processes form the basis of statistical learning theory. We have extended some of these inequalities to deal with vector-valued functions.

(2.14) **Theorem** (Hoeffding's Inequality): Let X be a set, D a probability distribution on X, and $f_1, \ldots, f_m$ real-valued functions defined on X, with $f_i : X \to [a_i, b_i]$ for $i = 1, \ldots, m$, where $a_i$and $b_i$are real numbers satisfying $a_i < b_i$.

$$\Pr\left(\left| \left(\frac{1}{m} \sum_{i=1}^{m} f_i(x_i) \right) - \left(\frac{1}{m} \sum_{i=1}^{m} \mathrm{E}_{x \sim D}\left[f_i(x) \right] \right) \right| \geq \epsilon \right) \quad \leq \quad 2 \exp\left(\frac{-2\epsilon^2 m^2}{\sum_{i=1}^{m} (b_i - a_i)^2} \right)$$

The preceding theorem is due to Hoeffding and appears in [3].

(2.15) **Theorem** (Chebyshev's Inequality from [3]): For a random variable $x \in \mathbb{R}$ with mean μ and variance σ^2:

$$P\left(|x - \mu| \geq t \right) \quad \leq \quad \frac{\sigma^2}{t^2}$$

for all $t > 0$.

We prove the following extension to Chebyshev's inequality when that $x \in \mathbb{R}^k$ and $k \geq 1$.

(2.16) **Theorem** (Chebyshev's Inequality in Multi-Variate Form): For a random variable $x \in \mathbb{R}^k$ with mean vector μ and diagonal and identical covariance, σ^2,

$$\begin{aligned} P\left(\|x - \mu\| \geq t \right) \quad &\leq \quad \frac{\mathrm{var}(x)}{t^2} \\ &\leq \quad \frac{k \sum_{i=1}^{k} p_i(1 - p_i)}{t^2} \end{aligned}$$

for all $t > 0$.

Proof: The more familiar form of Chebyshev's inequality, from Theorem 2.15, can be stated in measure-theoretic terms as:

$$\mu\left(\{x \in X \mid f(x)\} \geq t\right) \leq \frac{1}{g(t)} \int_X g \circ f \, d\mu$$

where μ is a measure, X is a measurable space, f is a measurable, non-negative function, and $t \in \mathbb{R} > 0$. The function $g(t)$ is a non-negative, non-decreasing function of the domain of f. We are concerned with the distance to the mean, so we substitute $\|x - \mu\|$ for $f(x)$ and let $g(x) = x^2$, which leads to the following:

$$\begin{aligned} P\left(\|x - \mu\| \geq t\right) &\leq \frac{var(x)}{t^2} \\ var(x) &= E\left[(x - \mu)^T (x - \mu)\right] \end{aligned}$$

where $x \in \mathbb{R}^n$, $\mu \in \mathbb{R}^n$, and $var(x)$ denotes the variance of x, which is the squared distance between the vector x and the mean.

Baxter applies the Chebyshev inequality in Theorem 2.15 to multi-task learning by estimating the variance of the expected loss as the variance of a binomial random variable [13]. For multi-objective bias learning, we can define an upper bound on the variance using the multi-nomial distribution.

$$\begin{aligned} var(X) &= E\left[(x - \mu)^T (x - \mu)\right] \\ &= E\left[\sum_{i=1}^{n} (x_i - \mu_i)^2\right] \end{aligned}$$

Since the summation in the expected value is a convex function, we can use Jensen's inequality [3] such that:

$$\begin{aligned} E\left[\sum_{i=1}^{n} (x_i - \mu_i)^2\right] &\leq \sum_{i=1}^{n} \left[E\left(x_i - \mu_i\right)^2\right] \\ &\leq \sum_{i=1}^{n} var_B(x_i) \\ &\leq \sum_{i=1}^{n} k p_i (1 - p_i) \end{aligned}$$

where k is the number of trials and p_i is the probability for the ith component of x. The Chebyshev inequality then becomes:

$$\begin{aligned} P\left(\|x - \mu\| \geq t\right) &\leq \frac{var(x)}{t^2} \\ &\leq \frac{k \sum_{i=1}^{n} p_i (1 - p_i)}{t^2} \end{aligned}$$

□

Single-Objective Multi-Task Learning

To understand the results for the multi-objective case, we first discuss the single-objective case. Maurer examined the sample complexity of multi-task learning within the common transformation model. The

transformation operator, T, is applied to each task and single-task learning is then applied to the transformed task, leading to the transformed function class $F = G \circ T$, where G is the class of functions before transformation. By further assuming that any loss function l used to evaluate the learner is defined as $l(f(X))$ and has a Lipschitz constant $\gamma \in (0, 1)$, the following bounds were shown to exist [58].

(2.17) **Theorem**: Let $\epsilon, \delta \in (0, 1)$ and $f \in F$ be a class of functions. With probability $> 1 - \delta$, for all $f \in F$ and $\gamma \in (0, 1)$:

$$\text{er}(\vec{h}_F) \quad \leq \quad \hat{\text{er}}_\gamma(\hat{f}) + \frac{1}{\gamma(1-\epsilon)} \text{E}\left[\hat{R}_n^m(F)(X)\right] + \sqrt{\frac{\ln\left(\frac{1}{\delta\gamma\epsilon}\right)}{2mn}}$$

and

$$\text{er}(\vec{h}_F) \quad \leq \quad \hat{\text{er}}_\gamma(\hat{f}) + \frac{1}{\gamma(1-\epsilon)} \hat{R}_n^m(F)(X) + \sqrt{\frac{9\ln\left(\frac{2}{\delta\gamma\epsilon}\right)}{2mn}}$$

where the function $\text{er}(\cdot)$ refers to the error and $\hat{\text{er}}_\gamma$ is the empirical error obtained by using a function with Lipschitz constant γ.

Theorem 2.17 relies on the Rademacher complexity. The different bounds result from using the expected, as opposed to, empirical Rademacher complexity. Although the empirical complexity is more easily computed, it increases the bounds.

The following theorem is used extensively and generalizes Theorem 2.17 [58].

(2.18) **Theorem**: Let F be a $[0, 1]^m$-valued function class on $\mathcal{X}$ and let X a dataset of $\mathcal{X}$-valued independent random variables for fixed t and all the x_j^t are independently distributed. Fix $\delta > 0$, then:

$$\frac{1}{m}\sum_{t=1}^{m} \text{E}\left[f_t(X^t)\right] \quad \leq \quad \frac{1}{mn}\sum_{t=1}^{m}\sum_{j=1}^{n} f_t(x_j^t) + R_n^m(F) + \sqrt{\frac{\ln\left(\frac{1}{\delta}\right)}{2mn}}$$

and

$$\frac{1}{m}\sum_{t=1}^{m} \text{E}\left[f_t(X^t)\right] \quad \leq \quad \frac{1}{mn}\sum_{t=1}^{m}\sum_{j=1}^{n} f_t(x_j^t) + \hat{R}_n^m(F)(X) + \sqrt{\frac{9\ln\left(\frac{2}{\delta}\right)}{2mn}}$$

This theorem shows that the average of any function over-estimates the expected value depending on the Rademacher complexity and the required confidence level (δ).

2.2.2 Weighted Objectives

In order to prove the bounds for the weighted objective model, we must first bound the composite loss function in terms of its component loss functions. The following lemma shows that a weighted combination of Lipschitz functions is still Lipschitz but with a larger constant that depends on the weights. Combining this with Lemma 2.13 gives us the desired result.

(2.19) **Lemma**: Let $f : \mathbb{R}^n \to \mathbb{R}^k$, $\lambda \in \mathbb{R}^k$ and $\lambda_i \geq 0$ for all $1 \leq i \leq k$, and $x \in \mathbb{R}^n$ be such that

$$f(x) \quad = \quad \lambda^T \left(f_1(x), \ldots, f_k(x)\right)$$

where each f_i has a Lipschitz constant γ_i. Then the function f satisfies the following Lipschitz condition:

$$\|f(x) - f(y)\| \quad \leq \quad \|\lambda * \gamma\| \, \|x - y\|$$

for $x, y \in \mathbb{R}^n$, and $\|\cdot\|$ is the Euclidean norm.

Proof: We start by expanding the norm in the function values:

$$\begin{aligned} \|f(x) - f(y)\|^2 \quad &= \quad \sum_{i=1}^{k} |\lambda_i f_i(x) - \lambda_i f_i(y)|^2 \\ &= \quad \sum_{i=1}^{k} \lambda_i^2 \left|f_i(x) - f_i(y)\right|^2 \end{aligned}$$

Since each f_i satisfies the Lipschitz condition with constant γ_i, we have that:

$$|f_i(x) - f_i(y)|^2 \quad \leq \quad \gamma_i^2 \, \|x - y\|^2$$

Since all values are positive, we have that:

$$\begin{aligned} \|f(x) - f(y)\|^2 \quad &\leq \quad \sum_{i=1}^{k} \lambda_i^2 \gamma_i^2 \, \|x - y\|^2 \\ &\leq \quad \|x - y\|^2 \sum_{i=1}^{k} \lambda_i^2 \gamma_i^2 \end{aligned}$$

We define the component-wise product $*$, such that

$$\|\lambda * \gamma\|^2 \quad = \quad \sum_{i=1}^{k} \lambda_i^2 \gamma_i^2$$

Thus:

$$\begin{aligned} \|f(x) - f(y)\|^2 \quad &\leq \quad \left(\|\lambda * \gamma\| \, \|x - y\|\right)^2 \\ \|f(x) - f(y)\| \quad &\leq \quad \|\lambda * \gamma\| \, \|x - y\| \end{aligned}$$

□

(2.20) **Lemma**: Let $L = (l_1, \ldots, l_k)$ and $l_i : Y \times Y \to \mathbb{R}$ have Lipschitz constant γ_i, then:

$$\hat{R}_n^m(\phi \circ F) \quad \leq \quad \|\lambda * \gamma\| \, \hat{R}_n^m(F)$$

Proof: From Lemma 2.19, we know that the weighted loss function has a Lipschitz constant $\|w * \gamma\|$. We can then create a function ϕ with Lipschitz constant $\|w * \gamma\|$. Thus, from Theorem 2.13, we have the result above. □

Proof of Theorem 2.3

By Theorem 2.18, we have that:

$$\frac{1}{m}\sum_{l=1}^{m}\mathrm{E}\left[l_\lambda(x_j^t)\right] \quad \leq \quad \frac{1}{mn}\sum_{l=1}^{m}\sum_{i=1}^{n} l_\lambda(x_j^t) + R_n^m(\mathcal{L}) + \sqrt{\frac{\ln\left(\frac{1}{\delta}\right)}{2mn}}$$

because the theorem holds for any function defined on a dataset. From Lemma 2.20, we can relate the complexity of the loss function to the complexity of the error function, where

$$R_n^m(\mathcal{L}) \quad \leq \quad \|\lambda * \gamma\|\, \hat{R}_n^m(F)$$

□

2.2.3 Multiple Loss Functions

We present both one- and two-sided bounds for multiple loss functions. The one-sided bounds extend Theorem 2.17 to bound the worst-case sample complexity of Pareto-optimal learning the tradeoff surface. The two-sided bounds depend on Baxter's framework, which relies on covering numbers and earlier results from learning theory.

One-Sided Bounds

For the one-sided bounds, we again rely on the Lipschitz condition, showing that a vector-valued function in which each component is Lipschitz is itself Lipschitz. We then combine this with Lemma 2.20 to obtain the result.

(2.21) **Assumption**: Let the loss function $l : \mathbb{R} \to \mathbb{R}^k$ be such that

$$\begin{aligned} l &= (l_1, \ldots, l_k) \\ \|l_i(x) - l_i(y)\| &\leq \gamma_i \|x - y\| \end{aligned}$$

where $l_i : \mathbb{R} \to \mathbb{R}$ and $\gamma_i > 0$ for $1 \leq i \leq k$.

We prove the following lemma, relying on the preceding assumption, to derive the Rademacher complexity of the multi-valued loss function in terms of the hypothesis space.

(2.22) **Lemma**: Let $f : \mathbb{R}^n \to \mathbb{R}^k$ and $x \in \mathbb{R}^n$ be such that

$$f(x) \quad = \quad (f_1(x), \ldots, f_k(x))$$

where each f_i has a Lipschitz constant γ_i. Then the function f satisfies the following Lipschitz condition:

$$\|f(x) - f(y)\| \quad \leq \quad \|\gamma\|\, \|x - y\|$$

for $x, y \in \mathbb{R}^n$ and $\|\cdot\|$ is the Euclidean norm.

Proof: We start by expanding the norm with respect to the function values:

$$\|f(x) - f(y)\|^2 = \sum_{i=1}^{k} |f_i(x) - f_i(y)|^2$$

Since each f_i satisfies the Lipschitz condition with constant γ_i, we have that:

$$|f_i(x) - f_i(y)|^2 \leq \gamma_i^2 \|x - y\|^2$$

Since all values are positive, we have that:

$$\begin{aligned} \|f(x) - f(y)\|^2 &\leq \sum_{i=1}^{k} \gamma_i^2 \|x - y\|^2 \\ &\leq \|x - y\|^2 \sum_{i=1}^{k} \gamma_i^2 \end{aligned}$$

We note that

$$\|\gamma\|^2 = \sum_{i=1}^{k} \gamma_i^2$$

Thus:

$$\begin{aligned} \|f(x) - f(y)\|^2 &\leq (\|\gamma\| \|x - y\|)^2 \\ \|f(x) - f(y)\| &\leq \|\gamma\| \|x - y\| \end{aligned}$$

□

Thus, we assume that each loss function has a Lipschitz constant γ_i and that the loss functions are defined in vector form. We prove the following lemma which allows us to show that the Rademacher complexity can be extended to multi-valued loss functions.

(2.23) **Lemma**: Let ϕ be a function with Lipschitz constant $\|\gamma\|$, then:

$$\left\| \hat{R}_n^m(\phi \circ F) \right\| \leq \|\gamma\| \sqrt{k} |R_n^m(F)|$$

Proof: This follows from the Lipschitz constant derived in Lemma 2.22 and combined with Lemma 2.20.

□

By extending the Rademacher complexity to multiple objectives, we bound only the magnitude of the Rademacher variables. This avoids extending the notion of Rademacher complexity to vector-valued functions.

Proof of Theorem 2.4

From Lemma 2.23, we can prove the main result, upper bounds on the convergence error for multiple loss functions. By Theorem 2.18, we have that:

$$\frac{1}{m}\sum_{t=1}^{m}\mathrm{E}\left[L(h_t,X^t)\right] \quad \leq \quad \frac{1}{mn}\sum_{t=1}^{m}\sum_{j=1}^{n}L(h_t,x_j^t)+R_n^m(\mathcal{L})+\sqrt{\frac{\ln\left(\frac{1}{\delta}\right)}{2mn}}$$

because the theorem holds for any function defined on a dataset. From Lemma 2.23, we can relate the complexity of the loss function to the complexity of the error function, where

$$R_n^m(\mathcal{L}) \quad \leq \quad \|\gamma\|\,\hat{R}_n^m(F)$$

□

Two-Sided Bounds

The two-sided bounds follow Baxter's model of multi-task learning, where the algorithm optimizes loss functions mapping to $[0,1]$ [13]. Although not stated as two-sided bounds, recent work shows that the extension holds in the case of binary classification [14].

We start by defining a distance metric over vectors, whose properties will be used in later proofs [13].

(2.24) **Definition**: Let $x, y \in \mathbb{R}^k$ and $0 \leq \nu \leq 1$.

$$d_\nu\left[x,y\right] \quad = \quad \frac{\|x-y\|}{\|x\|+\|y\|+\nu}$$

The following properties can be shown to hold:

$$0 \leq \quad d_\nu\left[x,y\right] \quad \leq 1 \tag{2.25}$$

$$\frac{\|x-y\|}{\nu+2\sqrt{k}} \leq \quad d_\nu\left[x,y\right] \quad \leq \frac{\|x-y\|}{\nu} \tag{2.26}$$

Property 2.26 does not change the upper bound on the distance but does increase the lower bound. The lower bound is not used in the proofs.

(2.27) **Definition**: The *expected loss* of a hypothesis h over the probability distribution T_t over $(\mathcal{X}\times\mathcal{Y})$:

$$\mathrm{er}_{T_t}(h) \quad = \quad \mathrm{E}\left[l(h(x),y)\right]$$

We prove the following lemma, which is a symmetrization argument that bounds the expected error in terms of the difference between empirical estimates on two different datasets[3]. The idea is to show that the difference between the empirical and expected error is bounded by the difference between the empirical estimates of two independently sampled datasets. The main difference between this lemma and the one in [13] is the use of the multi-variate form of the Chebyshev inequality. This introduces the number of objectives, k, into the bounds on the sample size.

(2.28) **Lemma**[2]: Let $\mathcal{H}$ be a permissible set of functions from $(\mathcal{X}\times\mathcal{Y})^n \to [0,1]^k$ and let T_t be a probability measure on $(\mathcal{X}\times\mathcal{Y})^n$. For all $\nu > 0$, $0 < \alpha < 1$, and $m \geq \frac{2k}{\alpha^2\nu}$,

$$\mathrm{Pr}\left\{\mathbf{z}\in(\mathcal{X}\times\mathcal{Y})^{mn} : \sup_{\mathcal{H}} d_\nu\left[\hat{\mathrm{er}}_{\mathbf{z}}(h),\mathrm{er}_{T_t}(h)\right] > \alpha\right\} \quad \leq$$

$$2\mathrm{Pr}\left\{\mathbf{z}\in(\mathcal{X}\times\mathcal{Y})^{(2m,n)} : \sup_{\mathcal{H}} d_\nu\left[\hat{\mathrm{er}}_{\mathbf{z}(1)}(h),\hat{\mathrm{er}}_{\mathbf{z}(2)}(h)\right] > \frac{\alpha}{2}\right\}$$

[2]Extends Lemma 20 in [13].

Proof: By the triangle inequality for d_ν, if:

$$d_\nu \left[\hat{\text{er}}_{\mathbf{z}(1)}(h), \text{er}_P(h)\right] \quad > \quad \alpha$$

and

$$d_\nu \left[\hat{\text{er}}_{\mathbf{z}(2)}(h), \text{er}_P(h)\right] \quad < \quad \frac{\alpha}{2}$$

then

$$d_\nu \left[\hat{\text{er}}_{\mathbf{z}(1)}(h), \hat{\text{er}}_{\mathbf{z}(2)}(h)\right] \quad > \quad \frac{\alpha}{2}$$

By the multi-variate Chebyshev's inequality from Theorem 2.15, for any fixed h:

$$\begin{aligned}
\Pr\left\{\mathbf{z} \in (\mathcal{X} \times \mathcal{Y})^{(m,n)} : d_\nu \left[\hat{\text{er}}_{\mathbf{z}}(h), \text{er}_{T_t}(h)\right] < \frac{\alpha}{2}\right\} \quad &\geq \\
\Pr\left\{\mathbf{z} \in (\mathcal{X} \times \mathcal{Y})^{(m,n)} : \frac{\left|\hat{\text{er}}_{\mathbf{z}}(h) - \text{er}_{T_t}(h)\right|}{\nu} < \frac{\alpha}{2}\right\} \quad &\geq \\
1 - \frac{k \sum_{i=1}^{k} \text{er}_{T_t}\left(h^i\right)\left(1 - \text{er}_{T_t}\left(h^i\right)\right)}{m\nu\frac{\alpha^2}{4}} &
\end{aligned}$$

which is $\geq \frac{1}{2}$ as $m \geq \frac{2k}{\alpha^2\nu}$ and $\text{er}_{T_t}(h^i) \leq 1$, for all attributes $1 \leq i \leq k$.

□

We now bound the probability of a large deviation for two empirical estimates of the loss function. The previous result shows that the probability of the deviation of the empirical and expected losses are bounded by the probability for two empirical losses. We want to establish bounds for the empirical estimates, in order to bound the deviation between the empirical and expected values. We prove the following lemma, showing that the probability of a large deviation depends on the number of objectives.

(2.29) **Lemma**[3]: Let $\mathbf{f} : (\mathcal{X} \times \mathcal{Y})^n \to [0,1]^k$ be any function that can be written in the form $\mathbf{f} = f_1 \oplus \cdots \oplus f_n$. For any $\mathbf{z} \in (\mathcal{X} \times \mathcal{Y})^{(2m,n)}$,

$$\Pr\left\{\sigma \in \Gamma_{(2m,n)} : d_\nu \left[\hat{\text{er}}_{\mathbf{z}_\sigma(1)}(\mathbf{f}), \hat{\text{er}}_{\mathbf{z}_\sigma(2)}(\mathbf{f})\right] > \frac{\alpha}{4}\right\} \quad \leq \quad 2\exp\left(-\frac{\alpha^2\nu mn}{8\sqrt{k}}\right)$$

where $\sigma \in \Gamma_{(2m,n)}$ is chosen uniformly at random.

Proof: For any $\sigma \in \Gamma_{(2m,n)}$,

$$d_\nu \left[\hat{\text{er}}_{\mathbf{z}_\sigma(1)}(\mathbf{f}), \hat{\text{er}}_{\mathbf{z}_\sigma(2)}(\mathbf{f})\right] \quad = \quad \frac{\left\|\sum_{i=1}^{m}\sum_{j=1}^{n}\left[f_j(z_{\sigma(i,j)}) - f_j(z_{\sigma(m+i,j)})\right]\right\|}{\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\left\|f_j(z_{ij})\right\|}$$

Let $\beta_{ij} = f_j(z_{ij})$. For each pair ij, $1 \leq i \leq m$, $1 \leq j \leq n$, let Y_{ij} be an independent random variable such that $Y_{ij} = \|\beta_{ij} - \beta_{m+i,j}\|$ with probability $\frac{1}{2}$ and $Y_{ij} = -\|\beta_{ij} - \beta_{m+i,j}\|$ with probability $\frac{1}{2}$.

[3]Extends as Lemma 22 in [13].

$$\Pr\left\{\sigma \in \Gamma_{(2m,n)} : d_\nu\left[\hat{\text{er}}_{\mathbf{z}_\sigma(1)}(\mathbf{f}), \hat{\text{er}}_{\mathbf{z}_\sigma(2)}(\mathbf{f})\right] > \frac{\alpha}{4}\right\} =$$

$$\Pr\left\{\sigma \in \Gamma_{(2m,n)} : \left\|\sum_{i=1}^{m}\sum_{j=1}^{n}\left[f_j(z_{\sigma(i,j)}) - f_j(z_{\sigma(m+i,j)})\right]\right\| > \right.$$

$$\left.\frac{\alpha}{4}\left(\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\|f_j(z_{ij})\|\right)\right\} =$$

$$\Pr\left\{\left|\sum_{i=1}^{m}\sum_{j=1}^{n} Y_{ij}\right| > \frac{\alpha}{4}\left(\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\|\beta_{ij}\|\right)\right\}$$

Note that the mean of $Y_{ij} = 0$ and that the range for each Y_{ij} is

$$\left[-\left\|\beta_{ij} - \beta_{m+i,j}\right\|, \left\|\beta_{ij} - \beta_{m+i,j}\right\|\right]$$

Therefore, by Hoeffding's inequality (Theorem 2.14), we have

$$\Pr\left\{\left|\sum_{i=1}^{m}\sum_{j=1}^{n} Y_{ij}\right| > \frac{\alpha}{4}\left(\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\|\beta_{ij}\|\right)\right\} \leq$$

$$2\exp\left(-\frac{\alpha^2\left[\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\|\beta_{ij}\|\right]^2}{32\sum_{i=1}^{m}\sum_{j=1}^{n}\left\|\beta_{ij} - \beta_{m+i,j}\right\|^2}\right)$$

Let $\gamma = \frac{\sum_{i=1}^{2m}\sum_{j=1}^{n}\|\beta_{ij}\|}{\sqrt{k}}$. Since $0 \leq \frac{\beta_{ij}}{\sqrt{k}} \leq 1$, we have that $\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{\|\beta_{ij}-\beta_{m+i,j}\|^2}{k} \leq \gamma$. Thus,

$$2\exp\left(-\frac{\alpha^2\left[\nu mn + \sum_{i=1}^{2m}\sum_{j=1}^{n}\|\beta_{ij}\|\right]^2}{32\sum_{i=1}^{m}\sum_{j=1}^{n}\left\|\beta_{ij} - \beta_{m+i,j}\right\|^2}\right) \leq 2\exp\left(-\frac{\alpha^2\left(\nu mn + \gamma\sqrt{k}\right)^2}{32\gamma k}\right)$$

We note that $\frac{\left(\nu mn+\gamma\sqrt{k}\right)^2}{\gamma k}$ is minimized by setting $\gamma = \frac{\nu mn}{\sqrt{k}}$, which leads to:

$$2\exp\left(-\frac{\alpha^2\left(\nu mn + \gamma\sqrt{k}\right)^2}{32\gamma k}\right) = 2\exp\left(-\frac{\alpha^2\nu mn}{8\sqrt{k}}\right)$$

which completes the proof.

□

We prove the following theorem bounding the distance between the expected and empirical losses. The bounds on the distance function depends on a $\frac{\alpha\nu}{8}$-covering on the function spaces $\mathcal{H}$. In Lemma 2.29, we bounded the probability of a large divergence between two empirical estimates. Following a standard union-bound argument, the probability that the maximum such deviation is large is bounded on the probability than any such deviation is large. This is simply the union of several probabilistic events, whose probability is added across all events. The number of these events is given by the covering number, and represents the size of the set over which we are maximizing.

(2.30) **Theorem**: Let $\mathcal{H} \subseteq \mathcal{H}_1 \oplus \cdots \oplus \mathcal{H}_n$ be a permissible class of functions mapping $(\mathcal{X} \times \mathcal{Y})^n \to [0,1]^k$. Let z be an (n,m)-sample generated by sampling m times from $\mathcal{X} \times \mathcal{Y}$ according to T_t, where $m \geq \frac{2k}{\alpha^2 \nu}$. For all $\nu > 0$ and $0 < \alpha < 1$:

$$\Pr\left\{\mathbf{z} \in (\mathcal{X} \times \mathcal{Y})^{mn} : \sup_{\mathcal{H}} d_\nu\left[\hat{\text{er}}_{\mathbf{z}}(\mathcal{H}), \text{er}_{T_t}(\mathcal{H})\right] > \alpha\right\} \leq 4C\left(\frac{\alpha\nu}{8}, \mathcal{H}\right) \exp\left(-\frac{\alpha^2 \nu n m}{8\sqrt{k}}\right)$$

Proof of Theorem 2.5

We are now ready to prove the main result for two-sided convergence. We note that the result is a corollary of Theorem 2.30. In order to obtain bounds on the sample size of each task, m, we solve the right-hand side of the inequality in Theorem 2.30 for m.

Proof: Let $\delta = 4C\left(\frac{\alpha\nu}{8}, \mathcal{H}\right) \exp\left(-\frac{\alpha^2 \nu n m}{8\sqrt{k}}\right)$, solving for m yields the left side of the expression. The right side of the expression comes from Lemma 2.28, where $m \geq \frac{2k}{\alpha^2\nu}$ to bound the probability by $\frac{1}{2}$. The probability above is directly from Theorem 2.30.

2.3 Discussion

The sample complexity results we consider in this chapter generalize their respective counterparts in single-objective multi-task learning. Although increasing the number of objectives increases the sample complexity, we find that the increase is not substantial, being sub-linear in most cases. In addition, reducing the number of objectives to $k = 1$, reduces all results to their counterparts. Despite its increase in sample complexity, multi-task learning has several potential applications.

The data-dependent bounds we present are very similar to each other. Both rely on bounding the loss function, by weights or by component loss functions. The key difference is that the weights in the weighted version play a key role in the complexity. The vector-valued form could have a larger asymptotic complexity; however, the difference is not likely to be significant.

For the two-sided bounds, we find that the bounds extend nicely to the vector-valued case. The main distinction is that the covering is over a potentially larger space, requiring a larger covering. This tends to increase the sample complexity but not significantly.

3 Algorithms

We present algorithms for each of our proposed frameworks for multi-objective multi-task learning. Although each algorithm extends an existing single-objective algorithm, we show how to make them optimize conflicting objectives. We extend an existing multi-task SVM algorithm for each of the frameworks. We also show that in distance function learning, there is a tradeoff between the best-fitting distance function and the classification training error.

3.1 Designing an Algorithm

In designing a multi-objective multi-task learning algorithm, the main factor to be considered is what objectives to optimize. We consider three kinds of objectives: regularization, data-dependent loss functions, and clusters of tasks.

Regularization is common in machine learning in which we seek a hypothesis that minimizes loss and is inherently simple, with respect to some measure. This is the motivation behind regularization in support vector machines and model selection. It is straightforward to convert a regularized algorithm to a multi-objective formulation because the objectives tend to be in conflict. The tradeoff surface would aid in manually selecting the regularization parameters, augmenting existing cross-validation approaches. The disadvantage, however, is that there is really only one objective function that depends on the data. Consequently, we cannot compare the empirical and expected tradeoff surfaces. Since one of the objectives is defined over the hypothesis space, it may not be possible to compare empirical tradeoff surfaces of different algorithms.

Multi-objective learning algorithms with multiple data-dependent loss functions allow us to compare the expected and empirical tradeoff surfaces to determine convergence. We can also compare algorithms that optimize the same functions, allowing us to establish performance benchmarks. It may be difficult, however, to create several loss functions that can be minimized as part of the learning algorithm. It is also difficult to determine which loss functions to optimize, whether they are in conflict, and whether optimizing them even makes sense in the application domain.

Finally, assuming that we want to learn a single common hypothesis space for all tasks, clusters in the tasks can make this goal unachievable. A solution for one cluster may not be sufficient for another, or the presence of one cluster may increase the average loss in another. Instead, we can find a compromise between groups of tasks. The goal is to find one hypothesis space that minimizes the average loss in each cluster. The expected and empirical tradeoff surfaces can be compared, given the same clustering. When the clustering is not known in advance, the tradeoff surface may provide little insight.

3.2 Single-Task Learning

All of the multi-task algorithms in this chapter are based on a single-task learning algorithm. Support vector machines and nearest-neighbor algorithms are commonly used single-task learners. They are also easily extended to multiple tasks.

3.2.1 Support Vector Machine

The single-task formulation of the support vector machine algorithm, is a convex optimization problem [69]:

$$\begin{array}{rl} \min\limits_{w\in\mathbb{R}^q,\xi\in\mathbb{R}^n,b\in\mathbb{R}} & \frac{1}{2}\left\|w\right\|^2 + \frac{C}{n}\sum_{j=1}^{n}\xi_j \\ \text{subject to} & y_j(w^T x_j + b) \geq 1 - \xi_j \\ & \xi_j \geq 0 \end{array}$$

where q is the number of attributes of the example $x_j \in X$, n is the number of examples, $y_j \in \{-1, 1\}$ is the class label, ξ_j is a slack variable, and C is the regularization constant. The hypothesis of the SVM is the function $f(x) = w^T x + b$ and $\{x \mid wx + b = 0\}$ is the hyperplane that separates the classes by the largest margin, defined as $\frac{1}{\|w\|}$. The slack variables ξ_j denote the margin errors, when an example x_j does not lie inside the margin (satisfy the constraints). The objective function minimizes the complexity of the model, related to $\|w\|$ and the margin errors.

3.2.2 Nearest Neighbor

In the nearest-neighbor algorithm, a new example is labeled according to the label of its nearest neighbor [23, 43]. Given a dataset X and a distance function, d, the nearest neighbor of an example x is defined as:

$$NN(x, X) \quad = \quad \arg\min_{x'\in X} d(x, x')$$

where x need not be in X. The nearest-neighbor classification algorithm returns the following hypothesis:

$$h(x) \quad = \quad cl(NN(x, X))$$

Thus, the hypothesis requires knowledge of the training set X and the distance function d.

A commonly used distance function is the L_1 distance and its weighted extension.

(3.1) **Definition**: Given $a, b \in \mathbb{R}^q$, the L_1 *distance*, $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}^+$is defined as:

$$d(a, b) \quad = \quad \sum_{i=1}^{q} |a_i - b_i|$$

The L_1 is also known as the city-block or Manhattan distance. It is often preferred to the L_2 or Euclidean distance because it is easier to compute.

In the multi-task learning extension to the nearest-neighbor algorithm, the hypothesis space is characterized by the distance function. Specifically, weights for a weighted distance function are learned. The weighted version of the L_1 distance requires a weight vector $w \in \mathbb{R}^q$ with the following properties:

$$\begin{array}{rcl} w_r & \geq & 0 \\ \sum\limits_{r=1}^{q} w_r & = & 1 \end{array}$$

where $1 \leq r \leq q$.

(3.2) **Definition**: Given $a, b \in \mathbb{R}^q$ and $w \in \mathbb{R}^q$ as described above, the *weighted* L_1 *distance*, $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}^+$is defined as:

$$d(a, b) \quad = \quad \sum_{r=1}^{q} w_r \, |a_r - b_r|$$

The nearest-neighbor algorithm has many advantages and some disadvantages. Its primary advantage is that it has constant training time and no training error, perfect recall. In addition, it has been shown that the algorithm has, at worst, twice the Bayes (optimal) error for classification [28]. The disadvantage is that the evaluation time can become impractical for large datasets.

3.3 Weighted Objectives

We propose two algorithms for the weighted objective (regularization) framework. The first, SVM, extends an existing multi-task regularized learning algorithm to multiple objectives. The second, DFL, applies learns a distance function from multiple tasks, which is then used in the nearest neighbor algorithm.

3.3.1 Regularized Multi-Task Learning

The multi-task SVM algorithm follows the common model approach to multi-task learning. It finds a common weight vector that is added to the weight vector that a single-task SVM algorithm learns for each task. The algorithm was originally proposed for the single-objective (scalarized) case [33], and its extension to multiple objectives is shown here. It learns a different hyperplane for each task, composed of a common component v and a specialized component, w_t, for task T_t, $1 \leq t \leq n$, as follows [33]:

$$\begin{aligned} \min_{w \in \mathbb{R}^{q \times m}, v \in \mathbb{R}^q, \xi \in \mathbb{R}^{m \times n}} \quad & \lambda_1 \|v\|^2 + \tfrac{\lambda_2}{n} \textstyle\sum_{t=1}^{m} \left(\|w_t\| - \|v\|\right)^2 + \tfrac{C}{m} \sum_{t=1}^{n} \sum_{j=1}^{m} \xi_{t,j} \\ \text{subject to} \quad & y_j^t (x_j^t (w_t^T + v)) \geq 1 - \xi_{t,j} \\ & \xi_{t,j} \geq 0 \end{aligned} \tag{3.3}$$

where x_j^t is the jth example of the tth dataset and v is the common component of the model. Here the slack variables are defined for all examples in all datasets. The bias parameter b no longer appears in the formula (though it appears in the implementation) because it is simply an extension of the weight vector. The hypothesis for each task is $f^t(x) = (v + w_t)^T x + b$. The hypothesis space returned by this algorithm is determined by the weight vectors that can be added to v.

An important consideration in the original paper is the interpretation of the weights, specifically the ratio of weights $\frac{\lambda_1}{\lambda_2}$ [33]. The relation between the weights and the resulting model is better understood in the context of multi-objective optimization. A large ratio means that all tasks tend to have different models, the common component $\|v\|$ is minimized more. A small ratio means that models tend to have the same model, the deviation from the common component $\sum_{t=1}^{n} \left(\|w_t\| - \|v\|\right)^2$, is minimized more.

This is already a multi-objective algorithm, as are all support vector machine algorithms. From the discussion in Section 1.5.4, we recognize that Equation 3.3 is the scalarized form of the following multi-objective problem:

$$\begin{aligned} \min_{w \in \mathbb{R}^{q \times m}, v \in \mathbb{R}^q, \xi \in \mathbb{R}^{m \times n}} \quad & (f_1, f_2, f_3) \\ \text{subject to} \quad & y_j^t (x_j^t (w_t^T + v)) \geq 1 - \xi_{t,j} \\ & \xi_{t,j} \geq 0 \end{aligned} \tag{3.4}$$

where (f_1, f_2, f_3) are the objectives, defined as:

$$
\begin{aligned}
f_1(w, v, \xi) &= \|v\|^2 \\
f_2(w, v, \xi) &= \frac{1}{n} \sum_{t=1}^{n} (\|w_t\| - \|v\|)^2 \\
f_3(w, v, \xi) &= \frac{1}{m} \sum_{t=1}^{m} \sum_{j=1}^{n} \xi_{t,j}
\end{aligned}
$$

We see that the original problem can be reconstructed with weights $\lambda = (\lambda_1, \lambda_2, C)$. We can therefore interpret the multi-objective optimization problem as one of finding appropriate weight values for the scalar version of the problem. In the implementation, we restrict the weight vector to $\lambda = (\lambda, 1 - \lambda, C)$ where C is fixed. This allows us to apply efficient bi-criterion optimization algorithms to find the tradeoff surface for this problem. In the original paper, the ratio between the weights was determined via cross validation. Searching across λ has the same effect, except that we seek to find the tradeoff surface rather than a single solution.

Any solution obtained via the regularization approach is efficient. Since the problem is convex, any efficient point also has an appropriate weight vector. These results are summarized in the following theorem:

(3.5) **Theorem**: A solution (w, v, ξ) is a solution to Problem 3.3 if and only if it is a solution to Problem 3.4, provided that $\lambda_1, \lambda_2 > 0$.

Proof: Let (w, v, ξ) be an efficient solution to 3.4. Since the objectives are convex, Theorem 1.3 gives us that there is a weight vector λ such that the solution of the scalarized version of 3.4 is (w, v, ξ). Now, let (w, v, ξ) be an optimal solution of 3.3, for $\lambda_1, \lambda_2 > 0$. Let $\lambda = (\lambda_1, \lambda_2, C)$, then scalarizing 3.4 with λ results in an efficient solution, by Theorem 1.3. Since the scalarized objectives are equal, the solution to 3.3 has the same objective values as (w, v, ξ). Thus, (w, v, ξ) is a solution to 3.4.

□

The original algorithm was extended to nonlinear functions through kernels. This extension is left for future work. Our purpose here is to formulate the multi-objective problems.

In the multi-objective extension to regularization (SVM), there is only one data-dependent loss function. Thus, the empirical and expected tradeoff surfaces cannot be compared directly. Clearly, the $\|v\|$ and $\|w_t\| - \|v\|$ terms do not change as new examples arrive. The tradeoff surfaces can only be compared in each objective that depends directly on the data.

3.3.2 Distance Function Learning

The multi-task distance function learning algorithm (DFL) follows the common transformation approach to multi-task learning. Examples in all tasks are projected onto a diagonal projection matrix $W = diag(w)$ such that the entries are appropriate weights in a distance function, $w_i \geq 0$ and $\sum_i w_i = 1$.

The algorithm is an extension of an existing algorithm to multiple tasks and a weight vector [77]. The terminology is from the popular Relief-F algorithm, which is popular in distance function learning [50]. The extension to multiple tasks draws on our previous work in distance function for meta-learning, in which weights that led to better performance were retained [6, 30]. Thus, weights were reused to improve the convergence rate for related tasks. In this algorithm, we favor a more direct approach with a single weight vector for multiple tasks, illustrating the effect of multiple objectives. We seek a weight vector such that the weighted L_1 distance between examples in the same class is maximized and the distance between examples in different classes is minimized.

(3.6) **Definition**: The *component-wise distance* is defined as: $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^q$:

$$d(x, y) = (|x_1 - y_1|, \ldots, |x_q - y_q|)$$

The weighted L_1 distance can then be written as $d(x, y)^T w$ for a weight vector w.

(3.7) **Definition**: The *hit distance* is the distance to the nearest different example having the same class:

$$\begin{aligned} H_j &= d(x_j - x_h^{(j)}) \\ x_h^{(j)} &= \arg\min_{x \in X_h} \|x_j - x\| \\ X_h &= \{x \in X \mid cl(x_h) = cl(x_j) \wedge x_h \neq x_j\} \end{aligned}$$

Here, H is a matrix consisting of the component-wise distance from each example to its nearest hit. The hits and misses refer only to those examples in the same dataset.

(3.8) **Definition**: The *miss distance* is the distance to the nearest example in a different class:

$$\begin{aligned} M_j &= d(x_j, x_m^{(j)}) \\ x_m^{(j)} &= \arg\min_{x \in X_m} \|x_j - x\| \\ X_m &= \{x \in X \mid cl(x_m) \neq cl(x_j)\} \end{aligned}$$

What is of interest to us is the separation of the distance values. The miss distance should be as large as possible and the hit distance should be as small as possible. More importantly, for any given example the hit distance should be less than the miss distance. We pose this as a support vector learning problem, in which we seek to classify a distance vector into being a hit or miss. The weight vector learned by the SVM in this case serves as the weights for the distance function. The constraints can be expressed as follows:

$$\begin{aligned} M_j w + b &\geq 1 - \xi_j^{(M)} \\ H_j w + b &\leq -1 + \xi_j^{(H)} \end{aligned}$$

We assume that the distance values for the miss distance should be positive and the hit distance should be negative, assuming no sign restrictions on the weight vector. A solution to this SVM problem would separate the distances with a large margin. This may not be a good weight vector for a distance function, however. To see this, recall that the nearest-neighbor classifier returns the class label of the nearest example. That is, a good distance function should not only ensure that the hits are separated form the misses but it must also ensure that the nearest hit of each example is nearer than the nearest miss. This coupling of the examples is not considered by the SVM, in which the margin errors are indifferent to misses or hits. This can be incorporated by adding the constraints together as follows:

$$\begin{aligned} -H_j w - b &\geq 1 - \xi_j^{(H)} \\ M_j w + b - H_j w - b &\geq 2 - \xi_j^{(M)} - \xi_j^{(H)} \\ M_j w - H_j w &\geq 1 - \xi_j \end{aligned}$$

where we combine the slack variables $\xi_j = \xi_j^{(M)} + \xi_j^{(H)}$. This formulation may, however, result in a degenerate solution, namely $w = 0$. To avoid this situation, we introduce the distance gap:

$$\begin{aligned} o_j &= \frac{1}{q} \sum_{r=1}^{q} M_{j,r} - H_{j,r} \\ & 1 \leq j \leq nm \end{aligned}$$

which is a constant during optimization. The purpose of the distance gap is to require that the learned distance function be no worse than with equal weights. A "better" distance function should improve the distance gap. If the distances are not ideally separable, some examples will have a worse distance gap.

The following multi-objective formulation introduces a slack variable for distance gap and restricts the weights to the simplex. The optimization problem can be stated as:

$$\begin{gathered}\min_{w\in\mathbb{R}^n,\xi\in\mathbb{R}^{mn}} \quad \left(\frac{1}{mn}\sum_j H_j w, \frac{1}{mn}\sum_{j=1}^{mn}\xi_j\right) \\ M_j w - H_j w \geq o - \xi_j \\ \sum_{r=1}^{q} w_r = 1 \\ \xi_j \geq 0\end{gathered}$$

where w is a vector of weights shared by all tasks and ξ is a slack variable measuring how much each example violates the distance gap.

The constraints require that there is not a significant decrease in the distance gap as the weights change. The two objectives represent conflicting goals. First, we would like to make all examples with the same class have as small a distance as possible (minimizing $\sum_j H_j w$). Achieving this goal, however, could also reduce the miss distance $M_j w$, resulting in poor separability. Second, we can minimize the average slack. With a slack of 0, setting the weights to their original value would satisfy the constraints. These objectives result in a tradeoff between meeting the constraints and achieving a main objective.

Unlike the previous multi-task algorithm, this algorithm does not specialize for each task. Instead, it learns a common set of weights. To train an individual task, the 1-nearest-neighbor algorithm uses the learned weights in the L_1 distance.

This algorithm is an extension of existing work to multiple tasks and multiple objectives [77]. In the earlier form, the objective was to find a semi-definite matrix W such that

$$\begin{gathered}\text{minimize} \quad \sum_{ij}\eta_{ij}\left(x_i - x_j\right)^T W\left(x_i - x_j\right) + c\sum_{ij}\eta_{ij}\left(1 - y_{il}\right)\eta_{ijl} \\ -\left(x_i - x_l\right)^T W\left(x_i - x_l\right) - \left(x_i - x_j\right)^T W\left(x_i - x_j\right) \geq 1 - \xi_{ijl} \\ W \succeq 0 \\ \xi \geq 0\end{gathered}$$

where all x_i correspond to a single task, W is a semi-definite projection matrix, and η is a matrix denoting the nearest neighbors of each example. We can interpret the constraints as implementing a distance gap approach in the projected distance. The constant 1 is intended to resemble the margin constraint of the support vector machine. The objectives are also similar to those in our approach, except that in the original formulation, a regularization constant was applied to the slack variable term and discovered via cross validation. Our version of the algorithm above is a simplified with $W = diag(w)$ and $\sum w_j = 1$. With this modification, however, there is no guarantee that a margin of 1 is achievable. Often, this results in the degenerate solution with $w = 0$ and $\xi = 1$. The original distance values are always achievable with reasonable weights, however. We also use the L_1 distance as opposed to the L_2 distance in the original problem.

3.4 Multiple Loss Functions

Instead of balancing the complexity of the hypothesis space against the training error, we can balance multiple data-dependent loss functions directly. The loss functions we consider in this algorithm (SVMTPFP) are the losses in each class, averaged over all tasks. Specifically, we minimize the false-negative and false-

Labeled	Positive	Negative	Total
Positive	TP	FN	P
Negative	FP	TN	N
Totals	p	n	T

Table 3.1: A contingency table used to quantify the performance of a learning algorithm. Known class labels are row headers and predicted class labels are column headers (TP: true positives; FN: false negatives; FP: false positives; TN: true negatives).

positive rates, as would be computed from a contingency table shown in Table 3.1:

$$\begin{aligned} fp &= \frac{FP}{N} \\ fn &= \frac{FN}{P} \end{aligned}$$

In order to apply our optimization algorithms, we need a convex formulation of these loss functions. Figure 3.1 illustrates the different margin errors that can occur in binary SVM classification. In support vector classification, a training error occurs when a slack variable is positive, indicating one of the constraints cannot be satisfied. A false positive occurs when a negative example is labeled as positive, when it violates the margin constraints. Similarly, a false negative occurs when a positive example violates the margin constraints. However, an example x could violate the margin constraints but still be correctly labeled, if the sign is correct but $|wx + b| < 1$. We can bound the empirical fp and fn values by the slack variables. Let $\varphi(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$, then we have that

$$\begin{aligned} \text{FN} &\leq \sum \varphi(\xi_j^+) \\ \text{FP} &\leq \sum \varphi(\xi_j^-) \end{aligned}$$

where $\xi^{\pm}$ indicates the slack variables for examples belonging to the positive or negative class. Clearly, to minimize fn and fp for the training set, it suffices to minimize $\sum \xi_j^+$ and $\sum \xi_j^-$, respectively. From this inequality, we derive the following optimization problem, forming the SVMTPFP algorithm:

$$\begin{aligned} \min_{w \in \mathbb{R}^{q \times m}, v \in \mathbb{R}^q, \xi \in \mathbb{R}^{mn}} \quad & \tfrac{1}{m}\left(\tfrac{1}{n^+}\textstyle\sum_{j=1}^m \xi_{t,j}^+, \tfrac{1}{n^-}\sum_{j=1}^m \xi_{t,j}^-\right) && + \|v\|^2 + \frac{1}{n}\sum_{t=1}^n \left(\|w_t\| - \|v\|\right)^2 \\ \text{subject to} \quad & y_j^t\left(x_j^t(w_t + v) + b\right) && \geq 1 - \xi_{t,j} \end{aligned}$$

where ξ_j^+ is the slack for a positive example, n^+ is the number of positive examples, ξ_j^- is the slack for a negative example, and n^- is the number of negative examples. In this algorithm, we keep the regularization objectives fixed and separate the slack errors between the positive and negative classes.

3.5 Task Clusters

This algorithm, SVMGroup, follows the task cluster framework of multi-objective multi-task learning. The tasks are first partitioned into k clusters. Let g_i be the indices of the tasks in the ith cluster, where the clusters

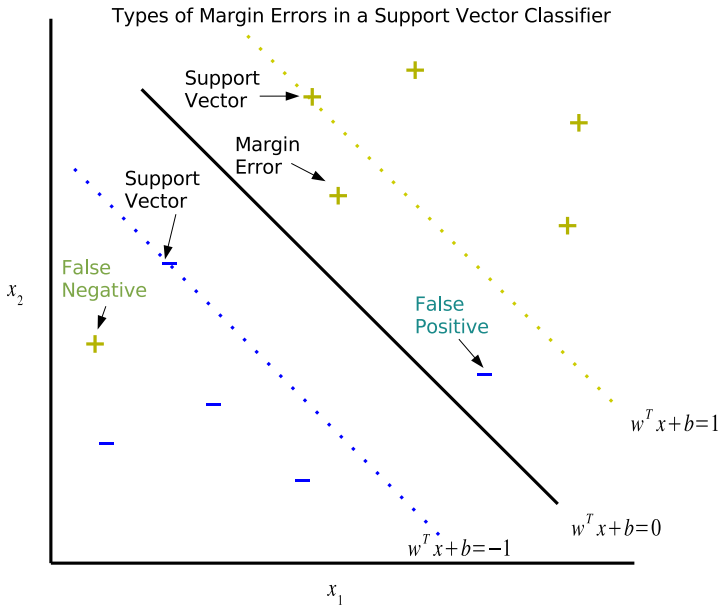


Figure 3.1: Types of margin errors that can occur in the support vector classifier. The SVMTPFP algorithm minimizes the occurrence of false positives and false negatives.

are mutually exclusive and exhaustive:

$$\begin{aligned} \bigcup_{i=1}^{k} g_i &= \{1, \ldots, k\} \\ g_i \cap g_j &= \emptyset \text{ for } i \neq j \end{aligned}$$

The objectives bound the average generalization ability for each cluster, as follows:

$$f_i(w, v) = \frac{1}{|g_i|} \sum_{t \in g_i} \|w_t + v\|$$

where $|g_i|$ is the size of cluster g_i and v is the common component in the multi-task SVM algorithm, which is shared by all tasks in all clusters. The component v makes this a multi-objective problem because there is only one common component and each cluster aims at being the only cluster, thus receiving all the influence of v. The optimization problem is defined as follows:

$$\begin{aligned} \min_{w \in \mathbb{R}^{q \times m}, v \in \mathbb{R}^q, \xi \in \mathbb{R}^{m \times n}} \quad & (f_1, \ldots, f_k) + \tfrac{C}{m} \textstyle\sum_{t=1}^{m} \sum_{j=1}^{n} \xi_{t,j} + \tfrac{1}{m} \sum_{t=1}^{m} \|w_t\| + \|v\| \\ \text{subject to} \quad & y_{t,i} \left(x_{t,i}(w_t + v) + b \right) \geq 1 - \xi_{t,i} \end{aligned}$$

where C is a constant, set to $C = 100$ in the experiments.

The purpose of this derivation of the objectives is to balance the generalization in each task rather than the more traditional objective of minimizing training error. A more intuitive formulation might be to let $f_i = \frac{1}{|g_i|} \sum_{t \in g_i} \sum_j \xi_{j,r}$, minimizing the margin errors in each cluster. This would seem to balance the error in each task, and it is very similar to our previously proposed algorithms. Unfortunately, these objectives are

not in conflict because the task-specific weights, w_t, are free parameters and always allow us to minimize the margin errors in all tasks. Since the only variable affected by all tasks is v, we can always minimize the margin errors by setting $v = 0$. In that case, we are just solving several single-task SVM learning problems at once. Instead, we minimize the generalization error in the cluster, which is proportional to the norm of each task's hyperplane, $w_t + v$. The additional terms bound the objective values when one of the objective weights is 0. Thus, we always want to minimize the margin error, but we also want our learned hypothesis spaces to minimize the generalization error in each cluster.

In this formulation, the margin error across all tasks and the norm of the common model have unequal weight. Our overall objective in SVM learning is to minimize the margin errors in each task. The difference between tasks is the portion of the weight vector w_t contained in the common component v. Regardless of the relationship between these weight vectors, the slack variables have the same weight, indicating our desire to minimize training error. What is in conflict across the different tasks is how well each task generalizes given the common component. Consider the relatedness of two tasks. If the two tasks are highly related, then the value of the objective weights, λ, does not matter because minimizing one objective minimizes the other. If the tasks are not related, then if we minimize the norm of one vector of weights and not the other ($\lambda = (1, 0)$), the common component will take most of the weight from the first task and the slack will be high for both tasks. Thus, there is a tradeoff between the norm of the weights across the different tasks.

The main advantage to this method is that it is now possible to consider the cost of each task separately without specifying a cost function at runtime. The disadvantage is that there are few existing algorithms to compute the tradeoff surface for multi-objective problem with more than 2 objectives. In the experiments, we will consider only 2 clusters of tasks.

The idea of learning for all clusters simultaneously was originally proposed our previous work in the context of meta-learning [5, 7]. There, we maximized a reward function for two disparate clusterings of tasks, modeled as collections of images. In contrast to the SVMGroup algorithm, the previous work used a distance function learning algorithm to find a weight vector for all clusters [6]. This work extends our previous work in that it directly utilizes multi-objective optimization.

3.5.1 Clustering Algorithm

This algorithm requires a clustering of the tasks, which is still the subject of ongoing research in the community and is outside the scope of the current work. Some related approaches are discussed in Section 5.2.2. We use an existing task clustering method based on a distance matrix [75]. Each dataset is represented as a feature vector consisting of the Euclidean distance between the mean of each class and the mean of the same class in all other datasets:

$$
\begin{aligned}
\hat{x}_t &= \left(d_+(X^t, X^1), d_-(X^t, X^1), \ldots, d_+(X^t, X^m), d_-(X^t, X^m)\right) \\
d_+(X^t, X^u) &= \left\| \frac{1}{|cl(X^t, +)|} \sum_{x \in cl(X^t, +)} x - \frac{1}{|cl(X^u, +)|} \sum_{x \in cl(X^u, +)} x \right\| \\
d_-(X^t, X^u) &= \left\| \frac{1}{|cl(X^t, -)|} \sum_{x \in cl(X^t, -)} x - \frac{1}{|cl(X^u, -)|} \sum_{x \in cl(X^u, -)} x \right\| \\
cl(X, \pm) &= \{x \mid x \in X \wedge cl(x) = \pm\}
\end{aligned}
$$

We then apply the k-means clustering algorithm to the resulting meta-dataset [60]. The returned clusters form the task clusters.

The clustering algorithm used in the implementation is unsupervised. Because the performance of this algorithm hinges on the clustering preprocessing step, we detail what an ideal clustering would be. This can

be embedded into a supervised clustering framework, such as in our previous work [6]. Ideally, two tasks should be in the same cluster if they have similar loss characteristics. For example, they can be learned adequately by the same hypothesis. Alternatively, the same hypothesis results in the similar loss values, even in the case of multiple loss functions. Drawing on existing work in defining task similarity, we can use an existing unsupervised clustering approach with the following distance function,

$$d(X_i, X_j) \quad = \quad L(h_i, X_j)$$

where the loss is with respect to the hypothesis trained on a different dataset. Thus, the unsupervised approach, minimizing the squared error across partitions results in tasks with similar loss values being grouped together. A fitness function for a supervised clustering algorithm could minimize the variance of the loss values of each task as follows,

$$\phi(g) \quad = \quad \sum_{i \in g} L(h_i, X_i) - \bar{L}$$

where $\bar{L}$ corresponds to the average loss in the cluster. The hypothesis used here, h_i, is the result of running a single-task learning algorithm on the dataset.

3.6 Implementation

All of these algorithms are implemented in Matlab and make use of existing optimization packages [56]. The Matlab functions are accessed via Java and some parts rely on Java code [40]. The Java framework is described in Chapter 6.

3.6.1 Software

The Java implementation is part of our new software library, which is described in Chapter 6. A simple philosophy guides the design of our classes, design classes based on how they are used. In the analysis phase of an experiment, we will apply a multi-task dataset and will also want to analyze the model for each individual task. The multi-task model is a PredictionModel, allowing us to apply a Dataset to it. It contains several PredictionModels corresponding to the single tasks. In evaluating a multi-objective multi-task model, we would like to evaluate each multi-task model. Thus, the multi-objective multi-task model contains multiple multi-task models.

The models, encapsulating the output of the algorithm, are defined in Java but the learner is Matlab function. The models, however, are reusable and do not depend on the Matlab implementation at all. The user can configure the Matlab options in the Factory class. The Factory then creates a generic learner that simply calls a specified Matlab function with the appropriate arguments. Integration with Matlab is seamless for the casual user. None of the serializable classes need access to Matlab, so checking the experiments and analysis the results can be done on a computer without Matlab. This is in contrast to most other frameworks where all the functionality of the learner would be tightly coupled with the Matlab implementation. An obvious limitation to our decoupling is that a Matlab error is only detectable at runtime which bypasses our extensive checking systems.

3.6.2 Convex Optimization

We use the CVX convex optimization library [41]. This library is a sophisticated wrapper around the existing SeDuMi optimization package [73]. The CVX library greatly simplifies the specification of optimization problems in Matlab.

Algorithm 1 CVX specification of the single-task SVM algorithm in Matlab using the CVX library, assuming the constant $\frac{C}{m} = 100$.

```
cvx_begin
    variables w(m) b e(n);
    dual variable s;
    minimize(0.5 * dot(w,w) + 100 * sum(e));
    subject to
       e == non-negative(n);
       y .* (X * w + b) >= 1 - e : s;
cvx_end
margin = 1 / norm(w,2);
```

Algorithm 2 Implementation in Matlab using the CVX library for the existing multi-task SVM algorithm, from [33].

```
function [uW, b, v] = multisvm_evgw(X, y, iTask, lambda1, lambda2)
  T = max(iTask(:));
  [m, n] = size(X);
  cvx_begin
    variables e(m) w(n, T) b(T) v(n);
    minimize(sum(e) + (lambda1 / T) * sum(norms(w)) + lambda2 * norm(v));
    subject to
      e == nonnegative(m);
      for i = 1:T
        iT = iTask == i;
        y(iT) .* (X(iT,:)  * (w(:,i) + v) + b(i)) >= 1 - e(iT);
      end
  cvx_end
  w = reshape(w, n, T);
  uW = w + repmat(v, 1, T);
end
```

The CVX library allows one to specify an optimization problem as one would write the formulas. For example, the source code that implements the one-task SVM algorithm is shown in Algorithm 1. Executing the `cvx_end` command solves the optimization problem. The declared variables are then assigned their optimal values. Since finding the support vectors requires the dual variables, we can declare which dual variables we want, s, and their values will be assigned upon completion of the optimization.

Although the package is designed for convex optimization, only a limited amount of convexity is allowed. The only usable solver, SeDuMi, is limited to semi-definite programming and has limited support for quadratic objectives and conic constraints [73]. In SVM algorithms, this limitation is not typically a problem.

Because the implementation of the algorithm almost exactly resembles the formulas, the implementation error is significantly reduced. For example, Algorithm 2 shows our implementation for the existing multi-task SVM algorithm from [33]. The remainder of the algorithms are defined similarly. The development process applied test-first development to Matlab, testing each algorithm on an average of 15 small test cases. Although there are few tools to evaluate the coverage of unit tests, upon manual inspection, the coverage rate is approximately 90%.

3.6.3 Setting the Objective Weights

Our algorithms return a set of efficient points, but we sometimes need to select just one. Numerous methods exist to reduce a multi-objective optimization problem to a single-objective problem, based on heuristics [29]. Here, we consider what is required to learn appropriate weights.

A popular method for selecting weights is cross-validation, in which we select several weights and take the one with best performance on a validation set. Selecting weights is a valid strategy because the algorithms we have presented are all convex optimization problems. Indeed, this approach was taken in the original papers.

There are several limitations to this approach. First, a space of more than a few weights quickly becomes infeasible, because the algorithm has to be trained and tested several times. Second, it is difficult to determine which solution is best for multiple objectives. Obviously, we can use prediction error as a standard objective, but there could be others. This would result in several equally optimal solutions, which would again result in a tradeoff surface. Finally, we lose the benefit of the tradeoff surface. We may be interested in the tradeoff surface in order to better understand the relationship between the objectives. Although cross-validation approach could be used in conjunction with the tradeoff surface, it is usually used to return a single answer.

In order to learn the weights, we must have some loss function defined over the objective values or over the weights. In practice, determining such a global objective function is difficult. In a tradeoff surface, the objective weights determine a hyperplane in the objective space. We could use the single-task SVM algorithm to learn weights given preference information for the objectives.

3.7 Evaluating Algorithms

Having multiple objectives complicates evaluating and comparing the results of our algorithms. The algorithms presented above require different evaluation methods. In some cases, existing single-objective approaches are appropriate, whereas in others, we must compare a set of evaluation measures or tradeoff surfaces.

3.7.1 Average Performance

Most existing work in multi-task learning evaluates an algorithms based on its average performance across all tasks. Although this is not entirely applicable to multi-objective multi-task learning, it can be applied to the algorithms discussed in Sections 3.3.1 and 3.3.2. Since the performance measures to compare reduce to scalars, existing statistical significance tests are applicable.

We reduce the efficient set output by each algorithm to a single element and then compare the average across all tasks. Given the efficient sets $\Theta_E^{(A)}$ and $\Theta_E^{(B)}$, we apply each solution to a testing set, given some performance measure l. We then compare the single best overall solutions as follows:

$$L^{(A)} \quad = \quad \min \left\{ \frac{1}{m} \sum_{t=1}^{m} l(\theta, X_t) \mid \theta \in \Theta_E^{(A)} \right\}$$

where we assume that the performance measure l is defined with the same scale over all tasks and is to be minimized. Common metrics such as accuracy and mean-squared error are valid measures for l. This procedure is performed for each fold, yielding a paired set of values: $\left(L_i^{(A)}, L_i^{(B)}\right)$ where $1 \leq i \leq k$ and k is the number of trials. We can then apply standard statistical tests for machine learning, such as the paired t-test or the F-test [28, 65].

The obvious advantage to this approach is that existing tests can be used and presented just as in the single-objective case. In the context of multi-objective learning, however, there are serious disadvantages. If some tasks have much better performance over other tasks, this important information could be lost. It is unclear whether the statistical significance for the average implies significance for each task individually. If the loss function l were vector-valued, we would have a set of Pareto-optimal losses instead of one point.

3.7.2 Comparing Tradeoff Surfaces

We would like to directly compare the output of our algorithms, tradeoff surfaces. In order to compare the tradeoff surfaces of different algorithms, both need to optimize the same objectives. The tradeoff surfaces for algorithms such as those in Sections 3.3.1 and 3.3.2 cannot be directly compared because their objective functions are different. Although both minimize error, their model-dependent objectives cannot be directly compared.

For algorithms that optimize the same (or similar) objective functions, their tradeoff surfaces can be directly compared. Unlike scalar performance measures such as expected loss, there are few valid statistical tests for multi-dimensional surfaces. In addition, it is not always clear when one surface is better than another. We can compute set-based dominance measures or the distance between the surfaces.

Dominance Measures

Dominance measures are designed specifically for tradeoff surfaces. Most measures compare the output of an algorithm to the true tradeoff surface. In our work, however, the true surface is rarely known. This limits the measures we can use to binary measures.

The binary ϵ-dominance measure returns a measure of the degree to which points are dominated between two tradeoff surfaces [85].

(3.9) **Definition**: Let $S, T \subseteq \mathbb{R}^k$ be two tradeoff surfaces and $\epsilon > 0$, the *binary ϵ-dominance measure* is defined as:

$$I_\epsilon(S,T) \quad = \quad \min\left\{\epsilon \in \mathbb{R} \mid \forall_{b\in T}\exists_{a\in S} : a \overset{\epsilon}{\leq} b\right\}$$

where $\overset{\epsilon}{\leq}$ is the ϵ-dominance measure, defined as:

$$\epsilon a_i \quad \leq \quad b_i$$

for $a, b \in \mathbb{R}^k$ and $1 \leq i \leq k$.

This can be understood as the minimum ϵ value such that one $a \in S$ dominates all $b \in T$. All tradeoff surfaces under comparison are assumed to have the same range. The measure is not symmetric so it may be the case that $I_\epsilon(S,T) \neq I_\epsilon(T,S)$. In addition, the comparison may fail in that $I_\epsilon(S,T) = I_\epsilon(T,S) = 0$. In this degenerate case, the tradeoff surfaces are not comparable.

To apply this measure in practice, we can compare tradeoff surfaces from the same fold and average the results. In each fold of cross-validation the algorithms return a tradeoff surface. We can compute $I_\epsilon(L^{(A)}, L^{(B)})$ for each fold, resulting in a set of ϵ values. We can then test whether ϵ is different from 0 using a standard hypothesis test. The test typically assumes that the values are distributed according to the normal distribution.

The advantages to this method are that it is relatively simple to compute and provides an intuitive notion of the degree of dominance. The disadvantages are that it could fail and there do not appear to be any existing statistical tests defined for this measure. It is unclear whether the distributional assumption over the ϵ values is justified.

Algorithm 3 Computing the Matching distance for two ROC curves A and B.

```
D = {}
for a ∈ A:
    d_b = NEAREST-NEIGHBOR-DISTANCE(a, B)
    add d_b to D
return μ_D / σ_D
```

Distance Functions

We can compute the distance between the surfaces using the Hausdorff distance, described in Section 2.1, or the Matching distance [27].

The Matching distance is the average nearest-neighbor distance between two sets, A and B, shown in Algorithm 3. We first compute the distance $d(a, B)$ for each $a \in A$. The resulting set of distances D is assumed to have a Gaussian distribution with mean μ_D and standard deviation σ_D. The final distance value is the mean distance divided by the standard deviation.

To apply distance functions to the tradeoff surfaces, we compare each fold. For each fold in cross validation, we compute the distance between $L^{(A)}$ and $L^{(B)}$ using our distance function. This gives us a set of distance values d. We assume that the distance values are distributed according to a Gaussian distribution. We can then test the statistical hypothesis that the mean of this distribution is zero, $\mu_d = 0$, rejecting the hypothesis when the distances are significantly different from 0 for some confidence value, such as $\alpha = 0.05$.

The advantages of computing the distance are that the distance function always returns a valid value and that this can be applied even if the sets are not exactly tradeoff surfaces. The main disadvantage is that the distance alone cannot tell us whether one tradeoff surface is significantly better or worse than other, only different. We can apply a dominance measure first and use the distance to determine statistical significance.

3.7.3 Set of Performance Measures

In multi-objective learning, we must often compare the efficient sets of algorithms when the tradeoff surfaces cannot be directly compared and when the performance of each task cannot simply be averaged. In these situations, none of the previous evaluation approaches can be directly applied. Since applying multiple evaluation measures on the efficient set results in a multi-dimensional set that is not likely to be shaped like a tradeoff surface, set-based measures cannot be used directly.

We can compute the distance between the results sets using the previously described distance functions. Determining whether one efficient set is significantly better than another is even more difficult in this situation. First, some evaluation vectors may dominate each other in the efficient set. We remove all dominated points which would leave only the undominated vectors in the set. We can then apply set-based dominance measures to the remaining set, but it is likely that the sets will be incomparable. The distance functions should perform as expected, however.

The advantage to computing several evaluation measures is that they are defined independently of the objectives of the algorithm, allowing us to directly compare multiple algorithms. The disadvantage is that the comparison complicates performance statistical significance tests.

3.7.4 Comparing our Algorithms

The tradeoff surfaces of the proposed algorithms cannot be directly compared because the algorithms all optimize different objectives. The regularization objective in SVM does not depend on the dataset, so it is essentially only a single-objective algorithm. The standard evaluation measures are best suited to this algorithm. SVMTPFP minimizes two data-dependent loss functions, shown to approximate two performance measures. Any algorithm that minimizes these same objectives can be compared to SVMTPFP using any of the tradeoff surface comparison methods. SVMGroup has different objectives compared to all other methods. Although its objectives are model-dependent, they are proportional to the generalization error. Algorithms can be compared on a cluster-by-cluster basis using one of the standard performance measures. If another algorithm minimizes the error rate on the same clustering, then the tradeoff surfaces can be compared. DFL minimizes two objectives related to the nearest neighbor distance. The tradeoff surface of this algorithm cannot be directly compared to other algorithms unless the other algorithms' objectives are similar. The other evaluation methods can be used, but only after a single solution is selected.

For algorithms optimizing different objectives, we can compute performance measures on the efficient set. A problem arises. however, when comparing these sets of measures. The two algorithms may have different notions of efficiency. For example, SVM and DFL optimize very different objectives, but both algorithms expect the ideal solution to minimize error, to separate the classes. Thus, their notion of efficient solutions is compatible, although their objectives are not. The SVMTPFP algorithm, however, has a very different purpose. It is designed to explore the objective space of multiple data-dependent loss functions, neither of which is empirical error. If, for example, we compare algorithms based on the average true-positive rates, the SVMTPFP algorithm has values that span a large range of possible values. We would then conclude that SVMTPFP is inferior to SVM because sometimes the true positives are lower in SVMTPFP. A partial solution to this problem is to select a single efficient solution that represents the best solution in some way. This solution is used in the experiments.

4 Experiments

The purpose of these experiments is to validate the performance of the algorithms. For each algorithm, there is a specific goal. Generally, we want to verify that the objectives that we optimize are indeed in conflict and that the tradeoff surfaces are reasonably shaped. For the SVMTPFP algorithm, the purpose is to show that empirically efficient learning results in an expected tradeoff surface that is similar to the empirical tradeoff surface. For the SVMGroup algorithm, the purpose is to investigate the relationship between clusters of tasks learned by the same hypothesis space.

4.1 Evaluation Methods

This experiment evaluates each algorithm proposed in Chapter 3 to four multi-task datasets. We evaluate each algorithm alone because there are no existing multi-objective multi-task algorithms in the literature for comparison.

4.1.1 Tradeoff Surface

Since our algorithms have only two objectives, we can easily show the tradeoff surface. We can rewrite a multi-objective optimization problem with two objectives as a bi-criterion problem. The bi-criterion problem can be solved by applying a single objective solver several times. We rewrite the objectives as:

$$\min \quad \lambda f_1 + (1 - \lambda) f_2$$

where $\lambda \in [0, 1]$ is the single weighting factor. This restricts the objective space to the set of convex combinations of the objectives. If the objectives are convex and the constraints are convex, then each value of λ results in an efficient pair of objective values $(f_1^{(\lambda)}, f_2^{(\lambda)})$. Since λ has a restricted range, we can search linearly from $\lambda = 0$ to $\lambda = 1$. To generate the tradeoff surface, we run each optimization problem N times with λ equally spaced over the interval.

There are two key features that we want to see in the tradeoff surface figures. First, we do not want the tradeoff surfaces to converge to a point. Convergence here means that the objectives are not in conflict, so there would be no reason to apply multi-objective optimization. If the ideal point does not lie on the surface, we are assured that the objectives are in conflict and that there are tradeoffs between them. Second, we expect the surface to be convex and oriented toward the lower-left corner (the ideal point). This would imply that the bi-criterion method for sampling the weights traverses the range of objective values. If the tradeoff surface only consisted of two extremes, it would be unclear whether this is because the objectives are completely incompatible or whether there is some error in our weight sampling method.

4.1.2 Distance to the Tradeoff Surface

We expect the tradeoff surface to change as datasets and algorithms change. Since our algorithms optimize different objectives, their tradeoff surfaces cannot be directly compared with each other. For the artificial datasets we examine, the similarity of the tasks is determined by a hyper-parameter. We can see trends in the tradeoff surfaces by plotting the distance to a fixed tradeoff surface as a function of the hyper-parameter.

4.1.3 Accuracy of the Efficient Set

Two of our algorithms pose the regularization approach to optimization as a multi-objective problem. Although the optimization algorithm has two objectives, minimize loss and minimize complexity, there is really only one loss function defined over new examples. We evaluate the accuracy of each efficient hypothesis for our testing datasets.

The purpose of this evaluation is to verify that the accuracy on the testing set changes with the weights, that the algorithm is sensitive to the weight. The accuracy of each efficient hypothesis space is plotted versus the weight of the bi-criterion problem. The shape of these curves will vary with the dataset and algorithm. The curve also serves as a method to select a single solution by visual inspection. One can simply find the weight with the highest accuracy on a validation set and use that to select the weights for the objectives.

4.1.4 TPFP of Efficient Set

The SVMTPFP algorithm minimizes data-dependent loss functions. For algorithms like this, accuracy alone is not a sufficient indicator of performance. Instead, we show the true-positive and false-positive rates for each efficient hypothesis space.

Unlike the ROC curve, commonly used in single-task learning, the shape of these curves are not well-defined. For most of the algorithms, we do not even expect to see continuous curves but rather a set of points. The reason for this is that we expect the true- or false-positive rate is not a function of the weights for most algorithms. With ROC curves, we varied a parameter of an algorithm, the threshold, that was known to directly influence the values. For SVMTPFP, which directly minimizes the false-positive and false-negative rates, however, we expect to see continuous curves very much like ROC curves. The purpose of these figures is to show that we can design algorithms and evaluation methods for a specific set of loss functions in mind.

4.1.5 Average Accuracy

Most of the evaluation methods we have proposed thus far have no analogue in single-task or single-objective multi-task learning. Most existing multi-task learning works evaluate an algorithm by computing the average loss across all tasks and return a single value. Another common approach is to show the learning curve, the average loss as the number of examples in each task increases.

In multi-objective learning, however, the difficulty in applying these evaluation methods is in selecting a single efficient hypothesis space. Many heuristic approaches have been used [29], and we adopt a simple approach. Our approach is to use the hypothesis space that results in the maximum average accuracy across the testing set. Although not all the algorithms directly maximize the accuracy, it is a common denominator and easily computed. We compare the average accuracy of each algorithm on each dataset, and compute a standard t-test for statistical significance.

4.2 Results

Few benchmark datasets exist in the literature for comparing multi-task learning algorithms. It is therefore appropriate to discuss our results for each dataset separately. We discuss two synthetic datasets and two real datasets. The synthetic datasets allow us to control the expected relatedness between tasks, and each is suited to one particular algorithm.

A multi-task dataset is represented as a single dataset in the implementation. It has two meta-attributes, described in Chapter 6, one for the class label and one to indicate the task. An additional meta-attribute is used to store the predicted class label, so that the dataset is represented as follows:

$$(x, cl, cl, t)$$

where t is the task label, x is the feature vector, and cl are the class labels.

For each multi-task dataset, we generated a fixed-sized training set. The number of tasks was set to be 25 and the number of examples per task was controlled as follows:

$$\begin{aligned} n &\in \{50, 100, 150, 200, 250, 300, 350, 400\} \\ m &= 25 \end{aligned}$$

For the real datasets, 25 datasets were randomly selected and at most n examples were randomly selected for each task. In the event that the dataset contained less than n examples, all examples were used.

Each algorithm was trained on 10 random subsets of the multi-task dataset. The random subsets were selected through cross validation stratified with respect to the task label. This ensures that the algorithm has about the same number of examples from each task for training. All figures show the average across these subsets. Error bars are included when appropriate.

We designed and implemented a software framework in order to perform the experiments. The framework, described in Chapter 6, stores an experiment as a graph in which each node is a step to perform. In total, there were approximately 256,000 nodes in the graph. This resulted in 2,880 multi-task models being built. Each model takes approximately 5-10 minutes to run, due to the Matlab implementation and the overhead associated with reading and writing objects to and from Matlab. The experiment ran for approximately 5 days on a 1.4 GHz Pentium M CPU with 1024 MB RAM.

4.2.1 Synthetic Datasets

Independent Attributes

In the independent attribute dataset, the separability of the classes is controlled independent of the other attributes. Tasks are related if their attributes have similar separability.

Each task consists of two classes, each modeled as a single Gaussian. The models have distributions $N(\mu_+, I)$ and $N(\mu_-, I)$, respectively. For the positive class, we set $\mu_+ = (0, \ldots, 0)$ for convenience. For the negative class, the mean is defined as:

$$\begin{aligned} \mu_- &= (\phi(p_1), \ldots, \phi(p_q)) \\ p_r &\in [0, 1] \end{aligned}$$

where $1 \leq r \leq q$ is the ith attribute and $\phi(\cdot)$ is the inverse cumulative PDF function for the single-dimension normal distribution $N(0, 1)$. The displacement value is simply the quantile of the normal distribution. A displacement value of $p_r = 0.5$ results in $\phi(p_r) = 0$, which is the mean of the distribution. Thus, the separation between the classes, with respect to each attribute, depends on the distance between p_r and the non-separable value of 0.5. As $|p_r - 0.5|$ increases, the separability of the classes increases for attribute r.

The relatedness of multiple tasks is controlled by a single hyper-parameter σ. To generate multiple tasks, we select the p values from a hyper-distribution, $N(\mathbf{p}_0, \sigma I)$. Here the mean separation vector $\mathbf{p}_0$ is manually selected. The single hyper-parameter, σ, controls the variance in the separation values. For $\sigma \to 0$, the tasks become identical in that each has the same separation with respect to each attribute. Multi-task learning for this task distribution reduces to single-task learning. As $\sigma \to 1$, tasks have very different separation, and the common information is lost in the noise.

This dataset is designed to be used with the distance function learning method. If we interpret the learned weights as an indication of the utility of a feature for classification, then we would expect the learned weights to be proportional to the mean separation vector $\mathbf{p}_0$. The interpretation of the learned hyperplanes of an SVM is less clear, however.

The parameters used in the experiments are as follows. The hyper-parameter σ had the following values:

$$\sigma \in \{0.0, 0.1, 0.2\}$$

The initial feature map $\mathbf{p}_0$ was:

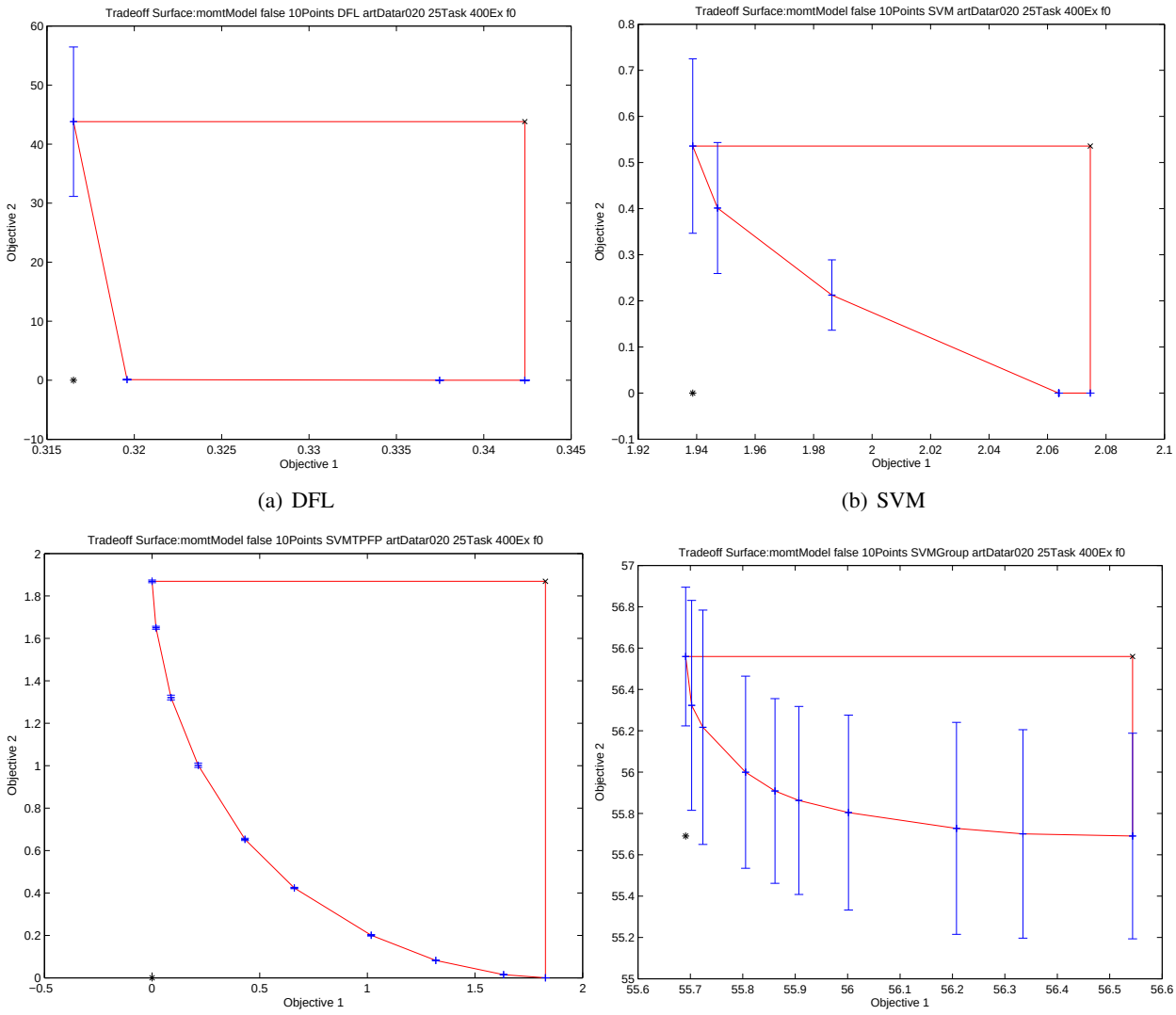
$$\mathbf{p}_0 = (0.4, 0.6, 0.5, 0.55, 0.45)$$

Tradeoff Surfaces Figure 4.1 shows the tradeoff surfaces for each algorithm on this dataset. As expected, most of the tradeoff surface have a convex shape, indicating conflicting objectives and a convex objective space. The overall shape of the curves does not change significantly as the sample size increases or across the different sub-datasets. In the figure, the DFL tradeoff surface has a very large value for the second objective, with high variance. This results from assigning 0 weight to the second objective $\sum \xi_i$, minimizing the hit distance only. The result of these objectives is the degenerate solution $w = 0$, which obviously minimizes all distance values but does not separate the classes. The SVM curves are almost linear and have about the same range in each objective. This means that the objectives have a nearly equal tradeoff ratio, decreasing one by δ decreases the other by $\approx \delta$. The SVMTPFP curves show that we can control the full range of error rates in each class in this dataset. Points on the curve typically have low standard deviation. SVMGroup shows about the same range in both objectives, indicating that the clusters have a symmetric loss relationship. This is to be expected because there is no inherent clustering in the tasks. The high standard deviation is due to random initialization of the clustering algorithm.

Distance to Tradeoff Surface Two trends can be observed in the distance between the tradeoff surfaces. First, Figure 4.2 shows that the tradeoff surfaces tend to converge to the tradeoff surface corresponding to the largest sample size. The standard deviation tends to decrease as the number of examples increases, indicating convergence to the expected tradeoff surface. The SVM shows the largest overall standard deviation, due to the different norm values associated with a weight of 0 for each regularization term. Second, Figure 4.3 shows that the distance to the tradeoff surface tends to increase as the dissimilarity increases. As the datasets become more dissimilar, the expected loss across all tasks increases. This means that multi-task learning is less effective as tasks become less similar.

Accuracy of the Efficient Set The objective weights have an impact on the average testing accuracy of each algorithm, as shown in Figure 4.4. We see a general increase in average accuracy as the relatedness of the tasks increases. This suggests that multi-task learning improves performance when tasks are related. Overall, the SVM algorithms appear to outperform DFL. For DFL, there are small changes as the weight changes. The accuracy tends to decrease as the weight increases, suggesting that separating the distance between the classes is more useful than minimizing the hit distance for this dataset. Interestingly, DFL had the best performance on average for dataset $\sigma = 0$, identical tasks. For SVM, however, there was little significant change in the accuracy as the weight increased, leading to a mostly flat accuracy curve. This indicates that regularization plays only a small role in predicting the testing performance. SVMGroup shows relatively little change indicating that one cannot obtain better overall accuracy by focusing on one group over another. The SVMTPFP shows a significant range of accuracy values as the weights change. The peak accuracy is attained at $\lambda = 0.5$, indicating that the classes are balanced. That is, better accuracy is obtained by minimizing the false negative and false positive rates equally.

TPFP of the Efficient Set As shown in Figure 4.5, the efficient set results in a set of (fp, tp) pairs. The pattern of (tp, fp) pairs for this family of datasets appears consistent across sample sizes but not across different relatedness values. In most cases, the range is small, with the SVM points sets typically consisting



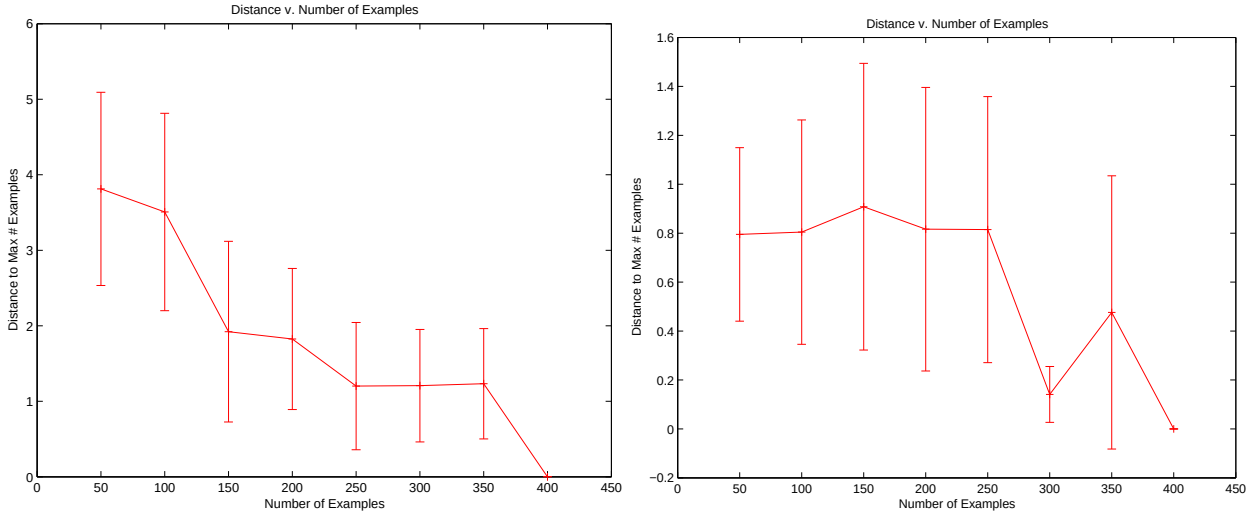
(a) DFL

(b) SVM

(c) SVMTPFP

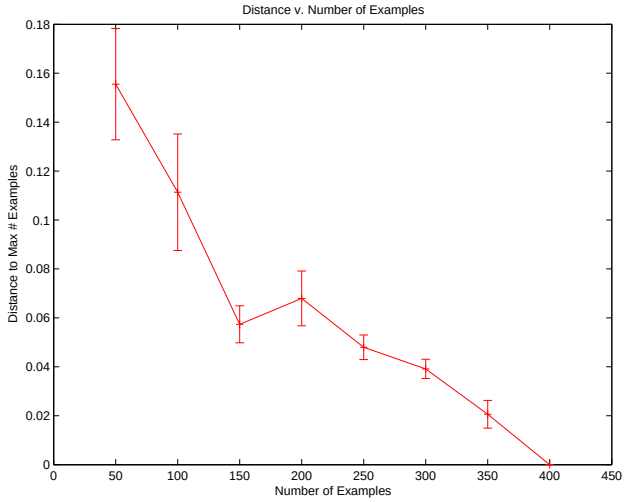
(d) SVMGroup

Figure 4.1: Tradeoff surfaces for the independent attributes dataset for 400 examples in each dataset and $\sigma = 0.2$. The horizontal axis shows the value for the first objective; the vertical axis shows the average and standard deviation for the second objective.

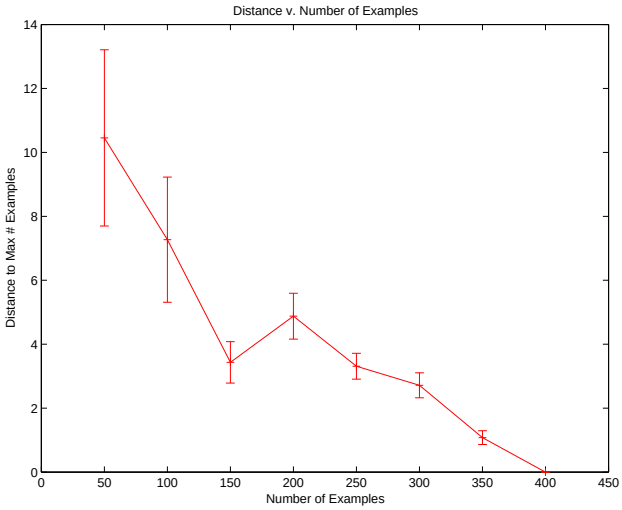


(a) DFL

(b) SVM

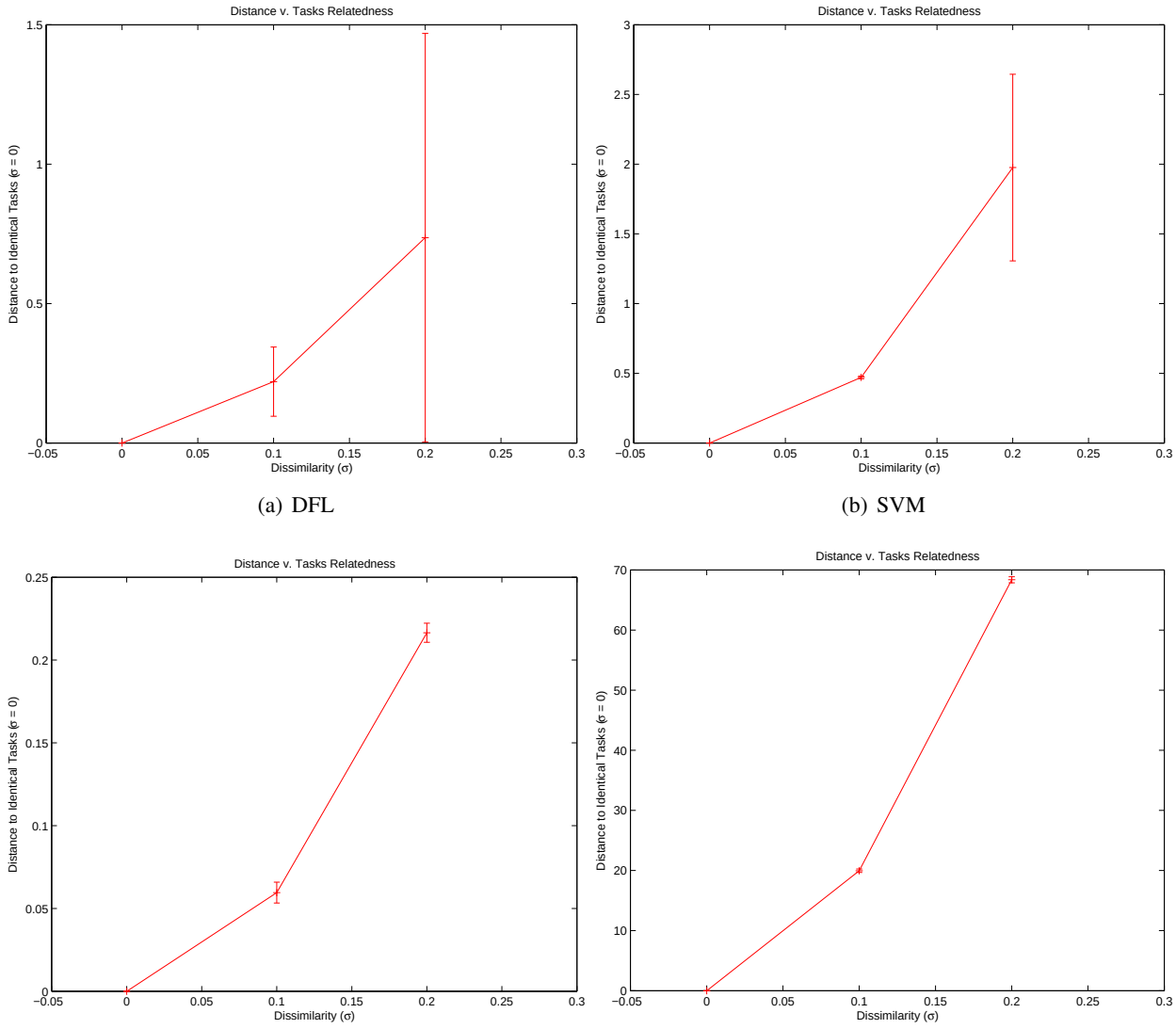


(c) SVMTPFP



(d) SVMGroup

Figure 4.2: Distance between tradeoff surfaces versus the number of examples in each dataset for the independent attributes dataset with $\sigma = 0.2$. The horizontal axis shows the number of examples in each task; the vertical axis shows the average and standard deviation of the distance to the tradeoff surface corresponding to 400 examples per task.



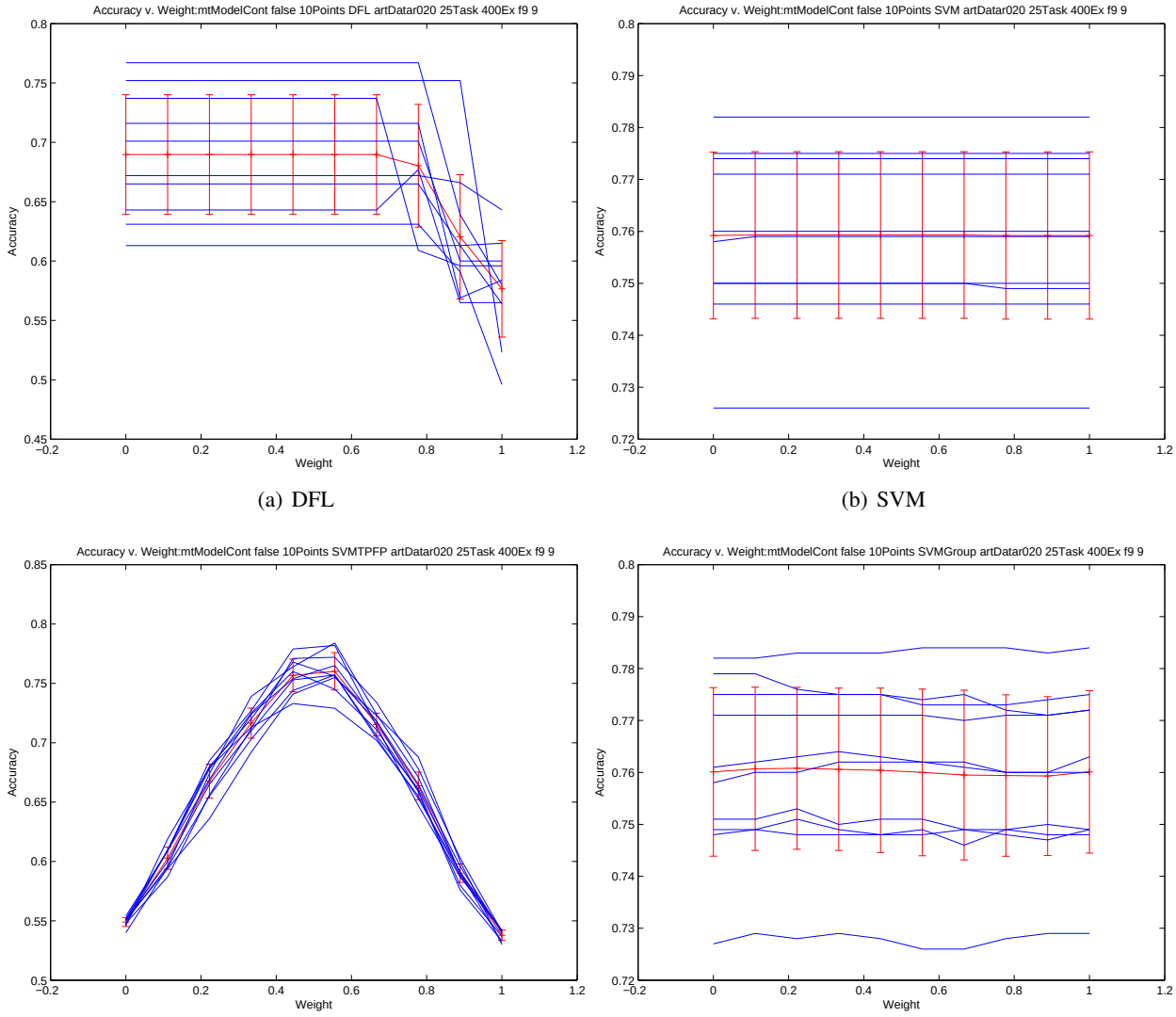
(a) DFL

(b) SVM

(c) SVMTPFP

(d) SVMGroup

Figure 4.3: Distance between tradeoff surfaces versus the relatedness of the datasets on the independent attributes dataset for 400 examples in each dataset. The horizontal axis shows the relatedness parameter σ; the vertical axis shows the average and standard deviation of the distance to the tradeoff surface corresponding to identical tasks ($\sigma = 0$).



(a) DFL

(b) SVM

(c) SVMTPFP

(d) SVMGroup

Figure 4.4: Accuracy v. weights on the independent attributes dataset for 400 examples in each dataset and $\sigma = 0.2$. The horizontal axis shows the weight of the first objective; the vertical axis shows the average and standard deviation of the accuracy for the 10 folds.

of a singleton. The SVMGroup patterns are different for each sample size, which is likely due to the random initialization of the clustering algorithm. The SVMFPTP algorithm appears to trace out a ROC curve with full range. The area under the curve appears to increase with the relatedness value.

Consumer Preferences

In the consumer preferences dataset, each consumer's preference for a set of products is a learning task. The dataset originated in marketing science literature, and has been used in related multi-task learning studies [33].

Each task is a consumer's preference for a set of products. The class labels indicate whether the consumer prefers or does not prefer a product. A consumer completes a survey consisting of 16 questions, each of which consists of 6 sub-questions. Each question q provides some information about 4 products: p_1, p_2, p_3, and p_4. The consumer is then asked to answer the following 6 (Yes or No) sub-questions:

- Do you prefer p_1 to p_2?
- Do you prefer p_1 to p_3?
- ...
- Do you prefer p_3 to p_4?

The consumer is modeled as a rational decision maker, so all possible comparisons are not necessary, which is why only 6 and not 12 sub-questions are asked. The questions result in a 6-example dataset, D_q, as follows:

$$\begin{aligned} D_q &= \begin{pmatrix} d_{1,2} & c_1 \\ d_{1,3} & c_2 \\ d_{1,4} & c_3 \\ d_{2,3} & c_4 \\ d_{2,4} & c_5 \\ d_{3,4} & c_6 \end{pmatrix} \\ d_{i,j} &= p_i - p_j \\ c_k &\in \{-1, 1\} \end{aligned}$$

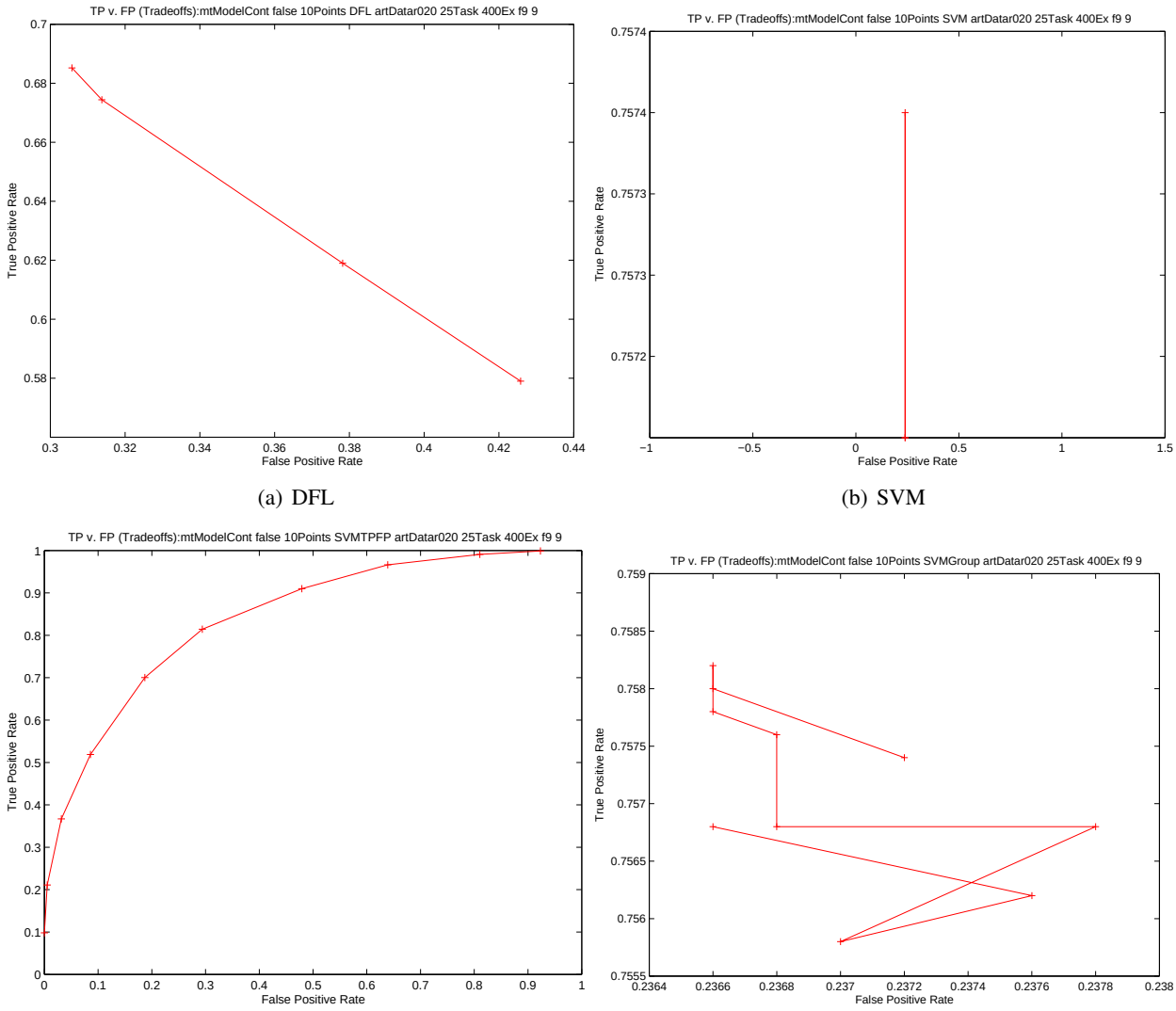
where p_i is a feature vector describing product i, $d_{i,j}$ is the distance vector between two products, and c_k indicates the answer to each sub-question in the survey.

To generate a single task, sets of 4 products are randomly generated for each question. Each product consists of a 4-dimensional ordinal vector in which each component indicates a particular attribute of the product, as follows:

$$\begin{aligned} p_i &= (o_1, \ldots, o_4) \\ o_i &\in \{1, 2, 3, 4\} \end{aligned}$$

where the ordinal values o_i indicate a rank for some feature, for example taste from bad to very good. For compatibility with SVM algorithms, the ordinal values are transformed into a vector of binary features:

$$\begin{aligned} p_i &= (b_1^1 b_2^1 b_3^1 b_4^1, \ldots, b_1^4 b_2^4 b_3^4 b_4^4) \\ b_j^i &= \begin{cases} 1 & o_i = j \\ 0 & o_i \neq j \end{cases} \end{aligned}$$



(a) DFL (b) SVM

(c) SVMTPFP (d) SVMGroup

Figure 4.5: (fp, tp) pairs for the efficient hypothesis spaces with 400 examples per dataset. The horizontal axis shows the average false positive rate; the vertical axis shows the average true positive rate, for 10 folds. A pair of points is connected by a line if the efficient points are connected along the tradeoff surface. Since each point represents a distinct efficient hypothesis space, these point sets should not be interpreted as ROC curves [34].

Since the final dataset consists of pairwise difference between products, the range of values is $\{-1, 0, 1\}$. The class labels indicate whether or not a consumer prefers one product versus another. The class values are generated by assuming that the consumer has a linear preference model. The class function $c(d_{i,j})$

$$\begin{aligned} c(d_{i,j}) &= \begin{cases} 1 & w^T d_{i,j} \geq 0 \\ 0 & w^T d_{i,j} < 0 \end{cases} \\ w &\in \mathbb{R}^{16} \end{aligned}$$

where w is the vector of weights corresponding to the difference between the binary features.

Each individual task consists of 96 examples of 16-dimensional binary vectors. To generate multiple tasks, the weight vector is selected according to the hyper-distribution:

$$\begin{aligned} w &\in N\left(\left(-\beta, -\frac{\beta}{3}, \frac{\beta}{3}, \beta\right), \ldots, \left(-\beta, -\frac{\beta}{3}, \frac{\beta}{3}, \beta\right), \sigma I\right) \\ \beta &\geq 0 \end{aligned}$$

where the parameter β controls the noise in the weights across all products and σ controls the similarity of tasks. For $\sigma \to 0$, consumers have very similar preferences since their underlying utility function is identical. For $\sigma \to \infty$, consumers have very different preferences. Typically, the parameters are related such that $\sigma = \gamma\beta$, where γ and β are defined by the user.

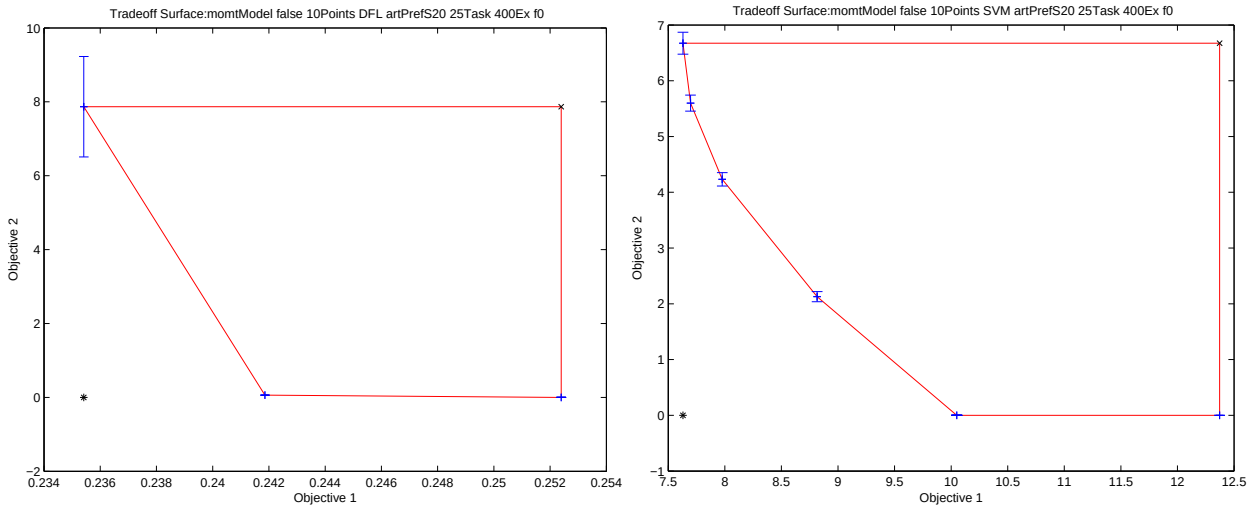
Since the class labels are determined by a linear function, this task is ideally suited to the SVM algorithm. We expect that as the γ value increases, the weights will be more specialized for each task with less in common.

In the experiments, we used the following parameters. The noise parameter, β, was set to $\beta = 2.0$. The relatedness hyper-parameter was set to:

$$\gamma \in \{0.5, 1.0, 2.0, 3.0\}$$

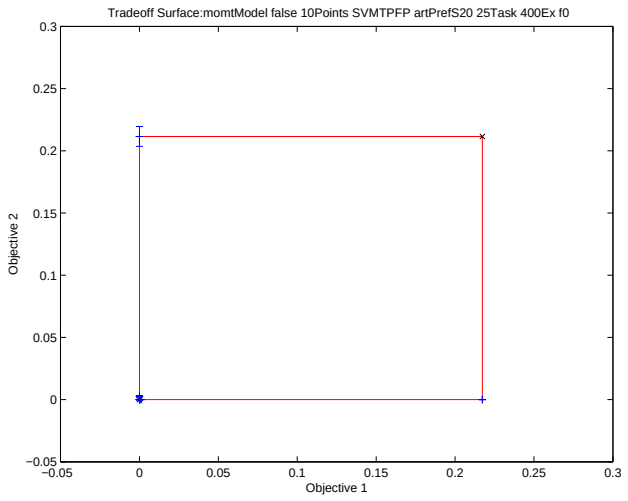
Tradeoff Surfaces Figure 4.6 shows the tradeoff surface for this family of datasets. Generally, the surfaces are convex as expected, and their shapes do not change as the sample size increases or over the datasets. The DFL tradeoff surface is concentrated in a small range for the first objective, indicating that there is little change in this objective. Most points have low standard deviation. SVM shows a large range for both objectives, with a larger range for the second objective than the first. This is due to the dimensionality (16-dimensions) and that the values have a large range in each of these dimensions. Most points have low standard deviation. The range of the second objective indicates the degree of commonality between the learned hypotheses for each task. As the similarity of the tasks decreases, with larger γ, we see that the range decreases because tasks have less in common. In the SVMTPFP we see that the objectives appear to be compatible, or at least capable of reaching their lowest values. The dense region of tradeoff values near the ideal point and the small range for both objectives show that the training error in each class can be made extremely low. For the SVMGroup, we see a large standard deviation for the points resulting from the random initialization of the clustering algorithm. The two objectives have the same range, indicating that the losses of the clusters are symmetric. This again indicates that there is no inherent clustering with the dataset.

Distance to Tradeoff Surface Figure 4.7 shows that the distance from the tradeoff surface to the set of identical tasks increases as the dissimilarity of the tasks increases. This is desirable because as tasks become dissimilar, we expect the algorithms to achieve worse loss values. Thus, the tradeoff surfaces should tend to dominate each other as tasks become more similar.

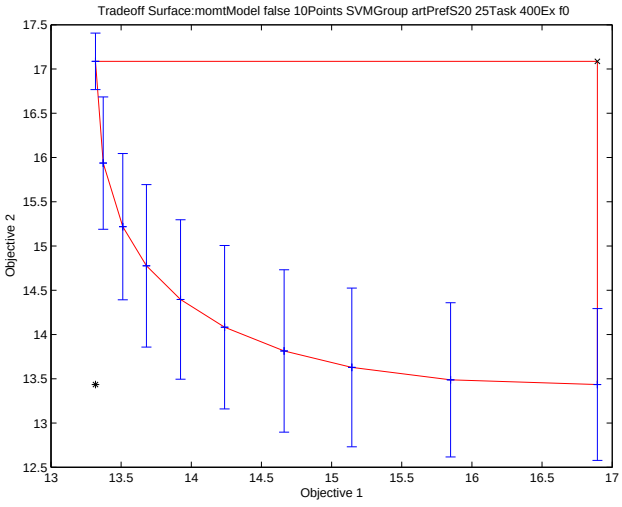


(a) DFL

(b) SVM

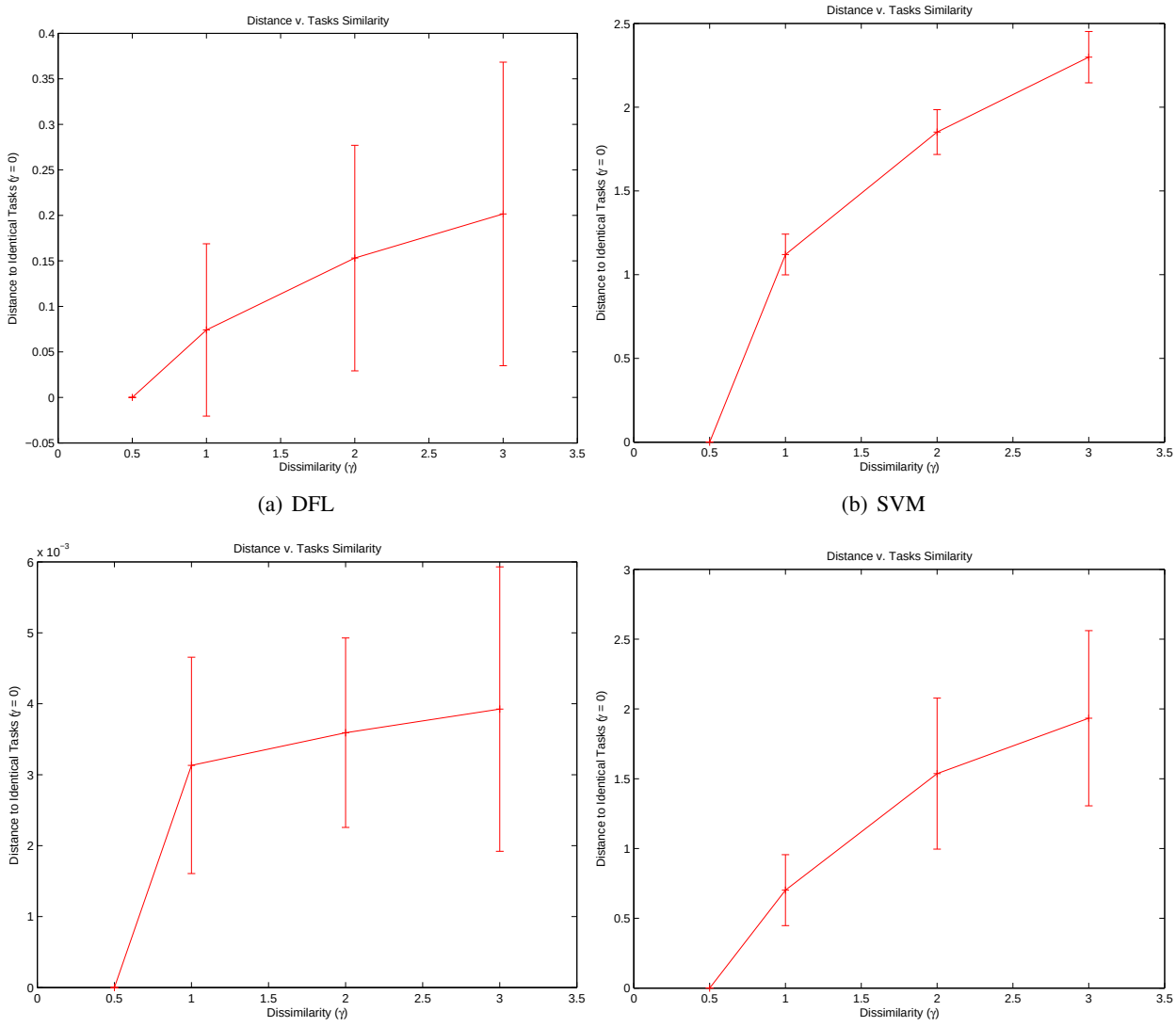


(c) SVMTPFP



(d) SVMGroup

Figure 4.6: Tradeoff surfaces on the consumer preferences dataset with 400 examples in each dataset and $\gamma = 2$. The horizontal axis shows the value for the first objective; the vertical axis shows the average and standard deviation for the second objective.



(a) DFL

(b) SVM

(c) SVMTPFP

(d) SVMGroup

Figure 4.7: Distance between tradeoff surfaces versus the relatedness of the datasets on the consumer preferences dataset with 400 examples in each dataset. The horizontal axis shows the relatedness parameter σ; the vertical axis shows the average and standard deviation of the distance to the tradeoff surface corresponding to nearly identical tasks ($\gamma = 0.5$).

Accuracy of the Efficient Set Figure 4.8 shows the accuracy values versus the objective weight for this family of datasets. Generally, the average accuracy is quite high. This is to be expected because the class relationship is linear. The average accuracy tends to increase as the relatedness of the datasets increases. The curves have a consistent shape across the relatedness values and the sample size. For DFL, the accuracy decreases significantly as the weight increases, suggesting that maximizing the separation between the classes is a better heuristic. The SVM curves have their highest values at the extremes of the weights, indicating that the tasks are related. SVMTPFP indicates that minimizing both loss functions with equal weight results in the highest accuracy. This indicates that the classes are well balanced. SVMGroup finds a different clustering of the datasets for each fold, but it does appear that concentrating on one cluster over another improves average accuracy. This is because several curves increase as a function of the weights and several decrease. If the clustering made no difference, all curves would be flat.

TPFP of the Efficient Set Figure 4.9 shows the (fp, tp) pairs for the efficient set for consumer preference dataset. Although there is no discernible pattern for most of the algorithms, some characteristics of the pattern are interesting. The DFL and SVM patterns are consistent as the sample size increases, suggesting that the same efficient set is output. This means that the algorithms are consistent. SVMTPFP shows what resembles the upper-left portion of the ROC curve (low FP, high TP). In contrast to a traditional ROC curve, only a small part of the range is explored. This is because the datasets are linear and well separated. SVMGroup appears to return random points, resulting from the random initialization of the clustering algorithm.

4.2.2 Real Datasets

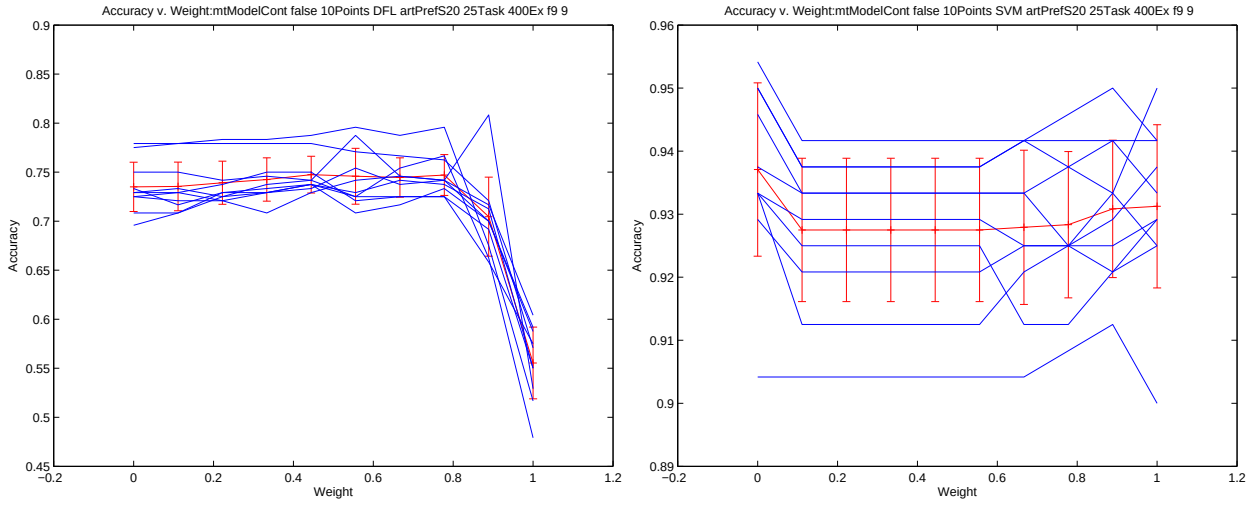
School

The school dataset consists of demographic information of over 15,000 students from 139 schools. The objective is to predict exam scores for a student, and each school is considered a learning task. Although the data have been used to study the effectiveness of schools, this dataset serves as a benchmark to test multi-task learning algorithms [8, 33].

Each task consists of features extracted from students at 139 secondary schools in London. The original dataset is a 50% sample. Each student is represented as a feature vector with attributes as shown in Table 4.1. On average, each school contains about 50 students, with over 200 students in some schools.

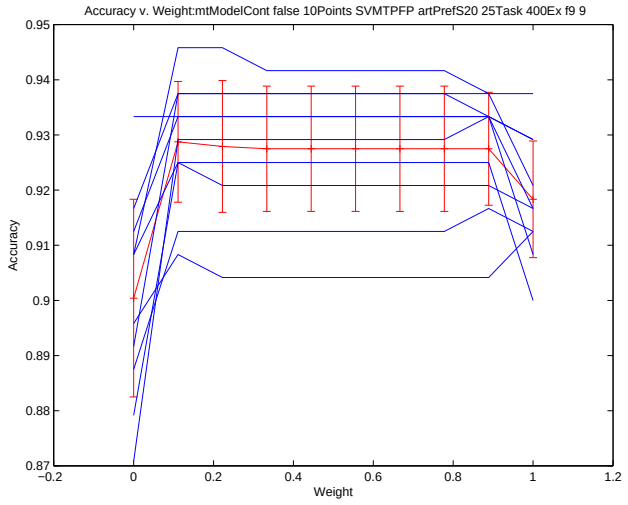
Since the algorithms described in Chapter 3 are designed for classification, the school dataset was transformed into a classification problem. Following related work, [33], the discrete attribute values (except school) were transformed into binary attributes. This yields a total of 27 attributes for each student. The numeric attributes were standardized to have mean 0 and standard deviation 1. The exam score attribute was transformed into a binary attribute with an approximately equal frequency for each value. This discretization reduces the numeric exam score to a pass or fail value. After converting the class to a binary feature, many students with identical feature values had the same class label. In both algorithms, duplicate examples do not change the training performance; thus, only unique feature values were retained. This reduces the number of examples from about 15,000 to about 5,000.

Tradeoff Surfaces Figure 4.10 shows the tradeoff surfaces for this dataset. Generally, the surfaces are convex and most algorithms have low standard deviation values for the most of the points. The shape of the curves does not change significantly as the sample size increases. The DFL algorithm results in a very small range of values, in contrast to previous datasets. The SVM shows a range twice as large for the second objective as the first. The SVMTPFP algorithm shows a smooth surface covering the full range of possible values. This indicates that the objectives are in conflict and are controllable. The SVMGroup

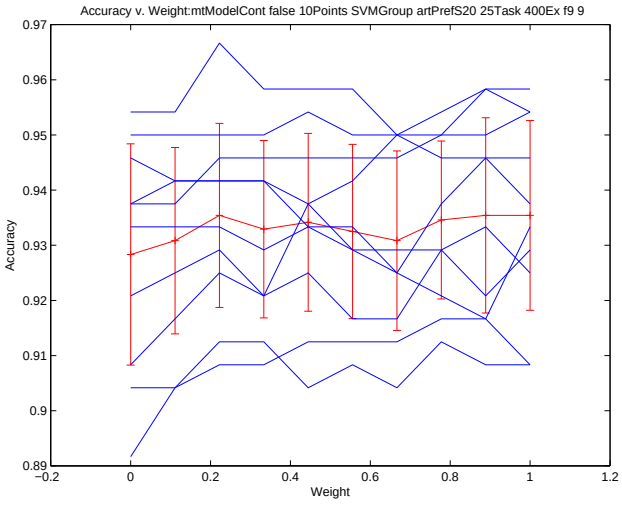


(a) DFL

(b) SVM

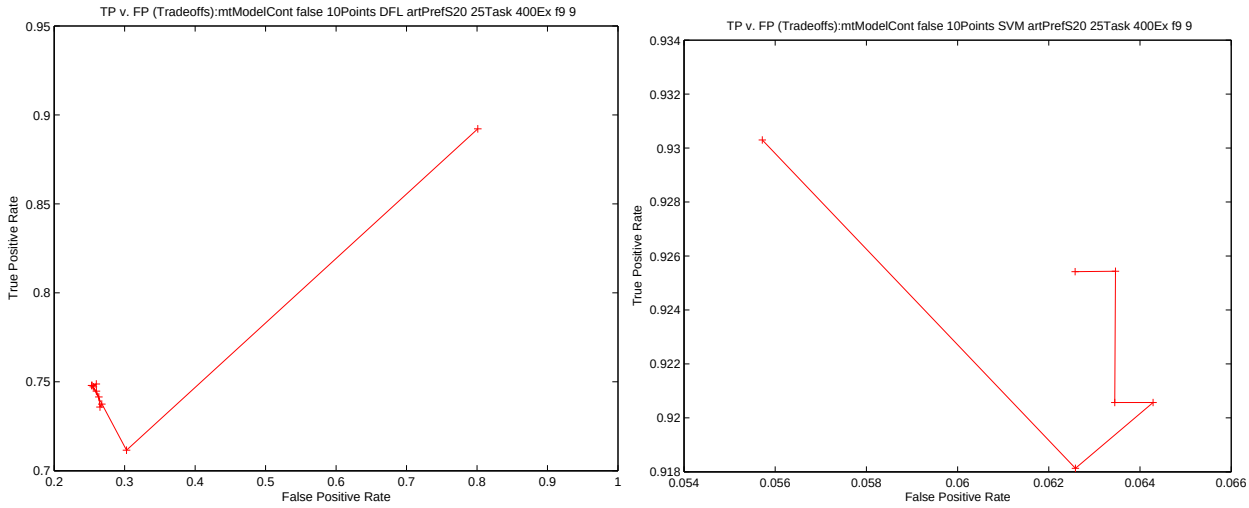


(c) SVMTPFP



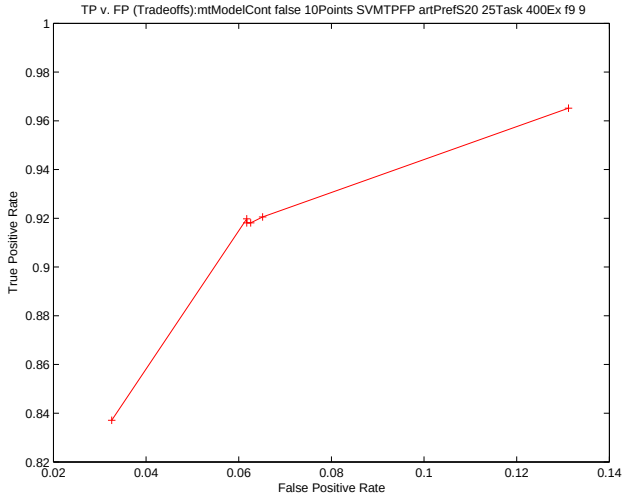
(d) SVMGroup

Figure 4.8: Accuracy v. weights on the consumer preferences dataset with 400 examples in each dataset. The horizontal axis shows the weight of the first objective; the vertical axis shows the average and standard deviation of the accuracy for the 10 folds.

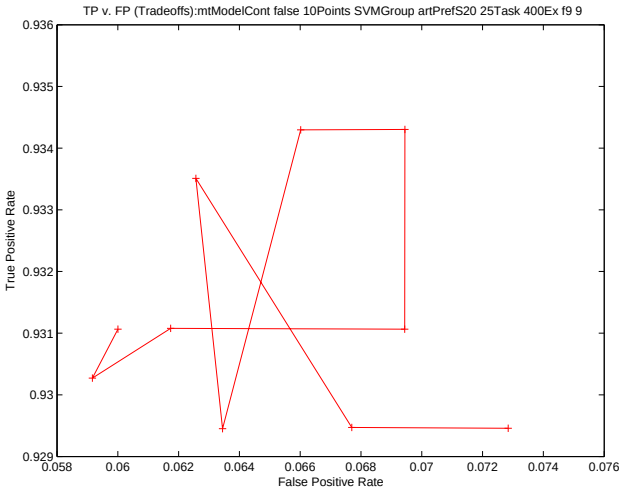


(a) DFL

(b) SVM



(c) SVMTPFP



(d) SVMGroup

Figure 4.9: (fp, tp) pairs for the efficient hypothesis spaces for the consumer preferences dataset with 400 examples per dataset and $\gamma = 2$. The horizontal axis shows the average false positive rate; the vertical axis shows the average true positive rate, for 10 folds. A pair of points is connected by a line if the efficient points are connected along the tradeoff surface. Since each point represents a distinct efficient hypothesis space, these point sets should not be interpreted as ROC curves [34].

Attribute	Value Range
Year	$\{1, 2, 3\}$
School	1-139
Exam Score	Numeric
FSM	$[0, 1]$
VR1 Band	$[0, 1]$
Gender	$\{0, 1\}$
VR Band	$\{1, 2, 3\}$
Ethnicity	$\{1, \ldots, 11\}$
School Gender	$\{1, 2, 3\}$
School Denomination	$\{1, 2, 3\}$

Table 4.1: Attribute information for the school dataset.

algorithm shows a restricted range with a large standard deviation for each point, due to randomization in the clustering algorithm.

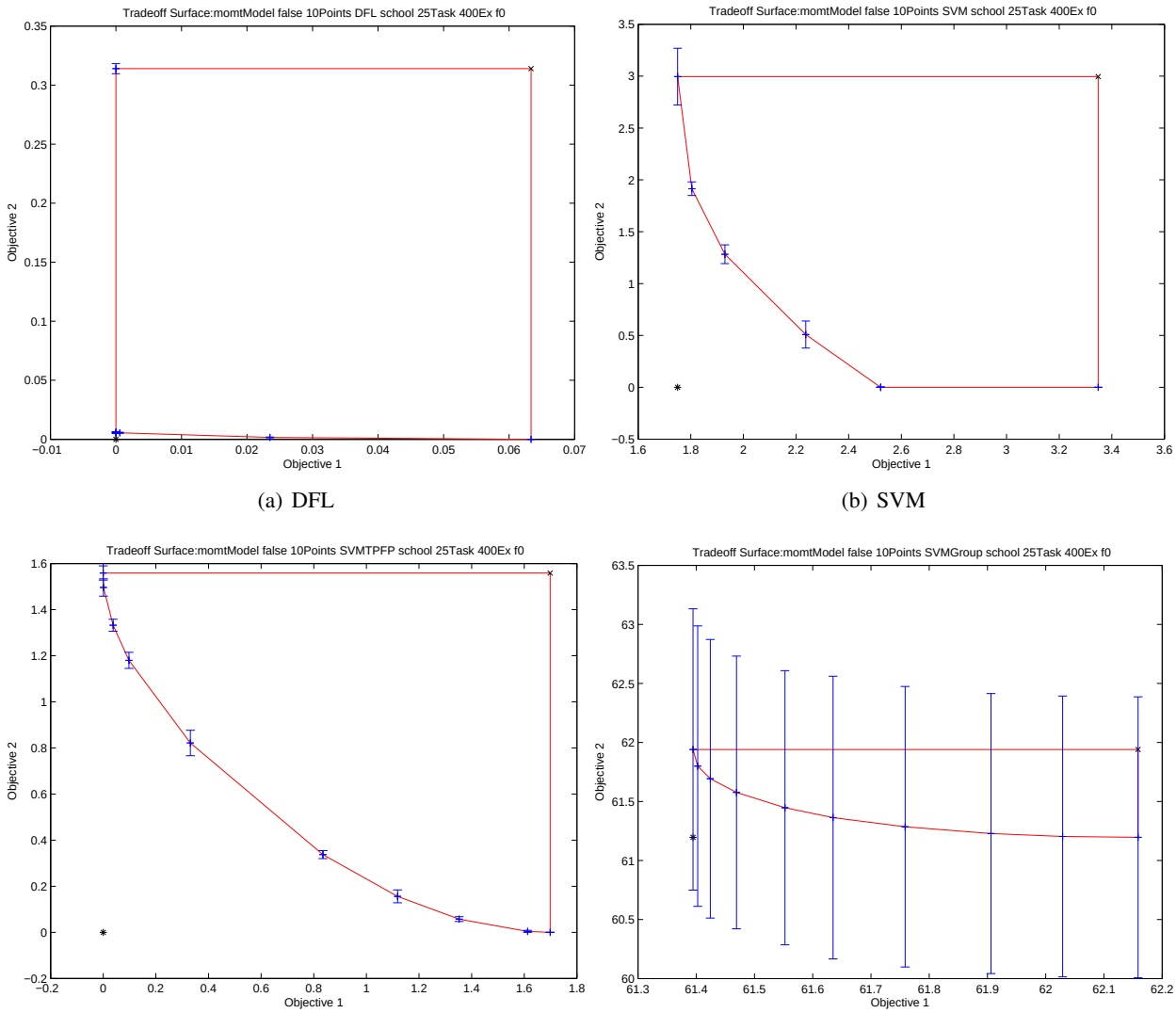
Accuracy of the Efficient Set As shown in Figure 4.11, none of the algorithms achieved more than 55% accuracy over any sample size. This suggests that the classification problem has high Bayes error. Despite the poor accuracy results, we can consider the trends of the accuracy versus the weights. Around $\lambda = 0.7$, the accuracy of DFL peaks, starting very low for $\lambda < 0.5$. This is the optimal weight balance, showing that minimizing the hit distance is a better heuristic than separating the classes for this dataset. The accuracy for SVM also tends to increase as the weight increases. This means that we can achieve better accuracy by specializing for each dataset separately. SVMTPFP shows that there are two distinct peaks, but the first peak is usually higher than the second. This means that we can obtain slightly better accuracy by emphasizing one class over another. Keeping the weights the same has a negative impact on the average accuracy. The curves for SVMGroup have a consistent shape as the number of examples increases. Emphasizing one group seems to increase average accuracy. Which group it is changes because of the random assignment of cluster labels.

TPFP of the Efficient Set Figure 4.12 shows the (fp, tp) pairs for the school dataset. The pattern for the DFL and SVM algorithms is consistent across sample sizes. All algorithms result in pairs that follow the random guess line ($fp = tp$), which is not surprising because the average accuracy is around 50%. SVMTPFP traces out a curve spanning most of the range. This line dominates the random guess line for most of the efficient set. The presence of the full range shows that the weights of SVMTPFP's objectives can control the false- and true- positive rates for this dataset.

Landmine

The Landmine dataset consists of 29 tasks with approximately 700 examples in each task. The tasks are assumed to belong to two groups, one corresponding to forest terrain and the other to desert. This dataset has been previously used in multi-task learning [80].

Each example is a component of a radar image, and the class label indicates the presence of a landmine. There is approximately a 6:1 class skew toward the no-landmine class. On average, there are less than 100 landmines in each task.



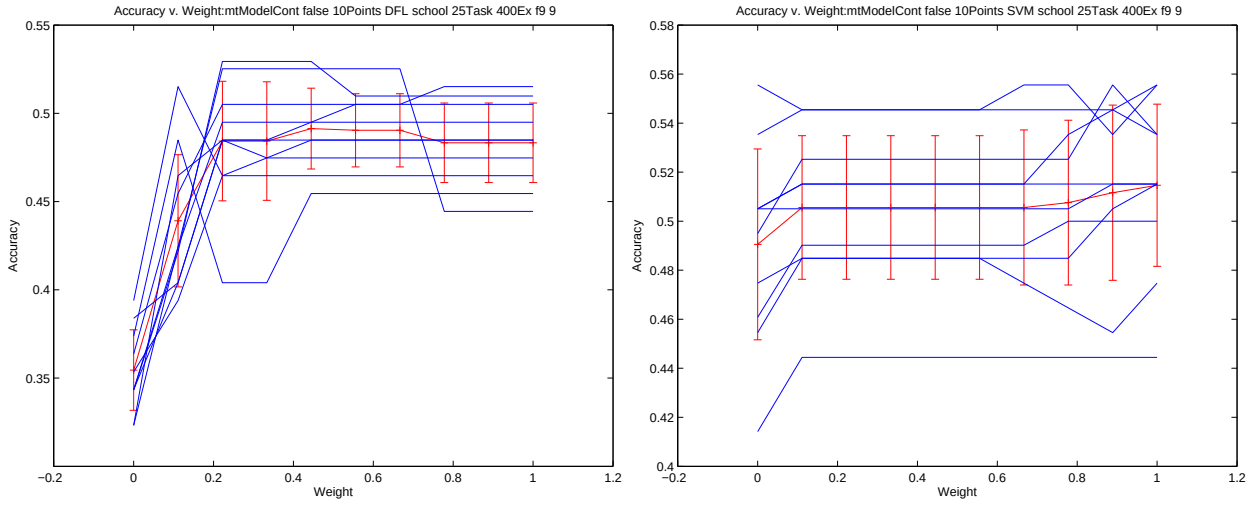
(a) DFL

(b) SVM

(c) SVMTPFP

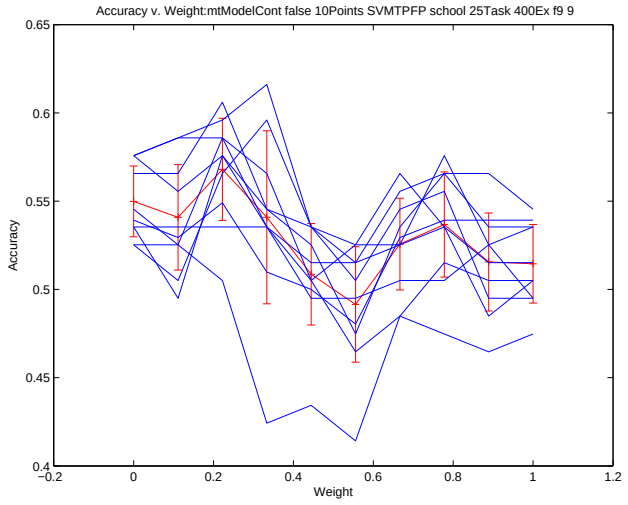
(d) SVMGroup

Figure 4.10: Tradeoff surfaces on the school dataset with 400 examples in each dataset. The horizontal axis shows the value for the first objective; the vertical axis shows the average and standard deviation for the second objective.

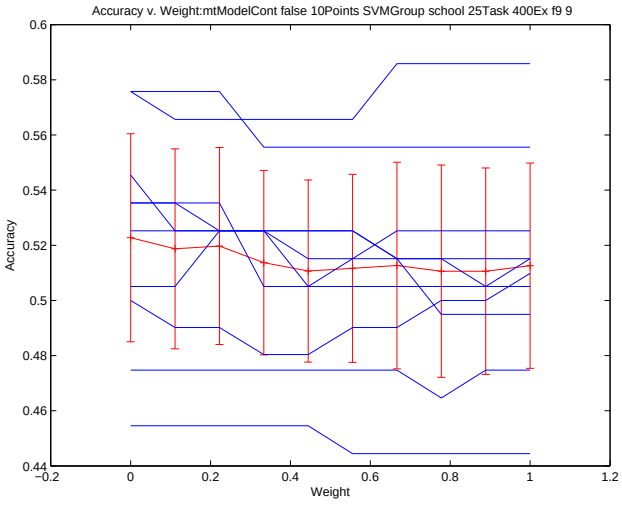


(a) DFL

(b) SVM

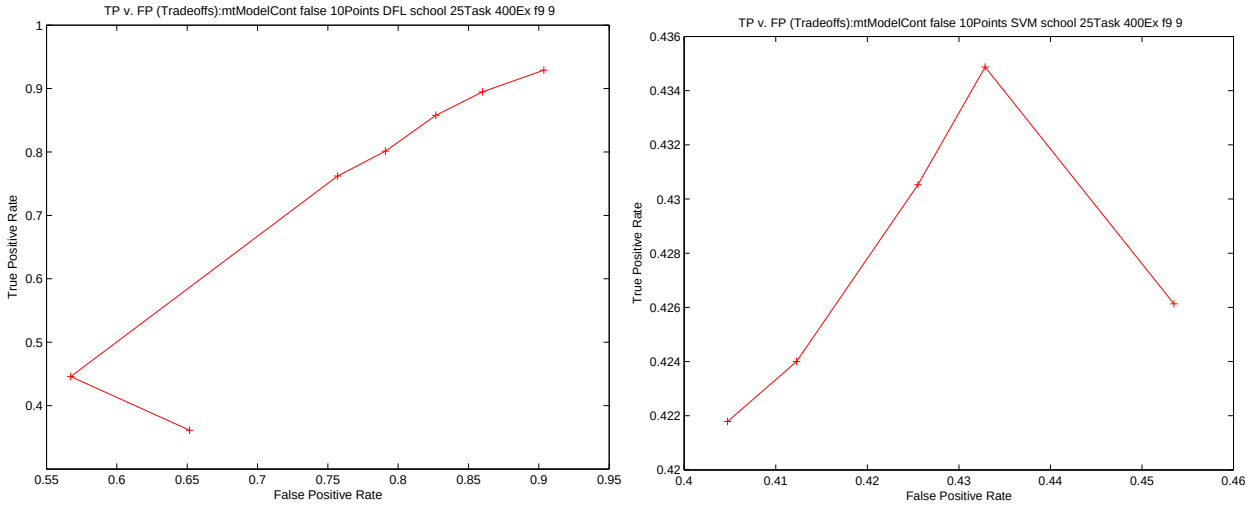


(c) SVMTPFP



(d) SVMGroup

Figure 4.11: Accuracy v. weights on the school dataset with 400 examples in each dataset. The horizontal axis shows the weight of the first objective; the vertical axis shows the average and standard deviation of the accuracy for the 10 folds.



(a) DFL (b) SVM

(c) SVMFPTP (d) SVMGroup

Figure 4.12: (fp, tp) pairs for the efficient hypothesis spaces for the school dataset with 400 examples in each dataset. The horizontal axis shows the average false positive rate; the vertical axis shows the average true positive rate, for 10 folds. A pair of points is connected by a line if the efficient points are connected along the tradeoff surface. Since each point represents a distinct efficient hypothesis space, these point sets should not be interpreted as ROC curves [34].

Each example consists of a 9-dimensional feature vector. Each feature is numeric-valued and was standardized to have mean 0 and standard deviation 1. The features convey radar information, such as the energy of the signal.

Because the class proportions are highly skewed, only AUC results are reported in the literature.

Tradeoff Surfaces Figure 4.13 shows the tradeoff surfaces for the landmine dataset. Most surfaces appear to be convex, indicating conflicting objectives. For DFL, we see that the objectives are significantly in conflict, an almost horizontal tradeoff surface. For SVM, the second objective value has an extremely small range, $\approx 2 \times 10^{-8}$, still within the precision of the optimization algorithm but still quite small. This suggests that the tasks have little in common, the norm of the common component of the hypotheses is always nearly zero. In contrast, most other datasets show some degree of commonality. A likely explanation for this is that the data is highly non-linear and is not adequately described by a linear function, since our algorithm is not kernelized. The SVMTPFP results in a smooth tradeoff surface showing the full range of values. This means that although the tasks may not share a common linear component, we can still control the TP and FP rates of the individual tasks. The SVMGroup surfaces show some tradeoff between groups with high standard deviation due to the random initialization of the clustering algorithms.

Accuracy of the Efficient Set Since there is a significant class imbalance, we expect the high average accuracy shown in Figure 4.14 because most of the examples belong to one class. For DFL, SVM, and SVMGroup, the accuracy curves are flat showing very small deviation from the average. The accuracy tends to decrease for DFL as the weight increases, indicating that separating the classes is a better heuristic for this dataset. The accuracy of SVMTPFP increases almost linearly with the weights, highlighting the class balance problem in that we can obtain high accuracy by simply assuming that everything is background. Although this sequence of tasks is believed to belong to two groups, there does not appear any advantage in overall accuracy in preferring one group to another.

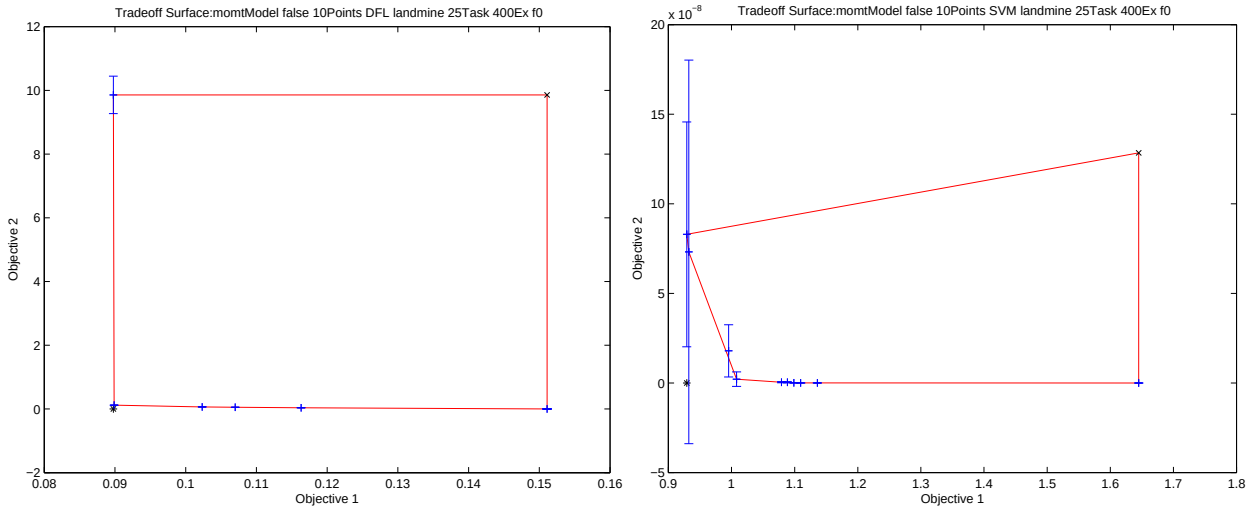
TPFP of the Efficient Set Figure 4.15 shows the (fp, tp) pairs for the landmine dataset. The points for the DFL and SVM algorithms are consistent across sample size. Both have a very small range consisting one line. SVMGroup again shows distinct patterns, but the patterns are contained in an extremely small range. SVMTPFP shows a curve very similar to a ROC curve. It indicates that the full range of values is possible.

4.3 Comparison

In general the goals for the experiment were achieved. All tradeoff surfaces have a convex shape, indicating that the objectives are in conflict and do not converge to a point. The tradeoff surfaces for the SVMTPFP algorithm indicate a full range of values in the false-negative and false-positive space. The TPFP plots confirm this resembling ROC curves in shape. The SVMGroup algorithm indicates a tradeoff between fitting different groups of tasks.

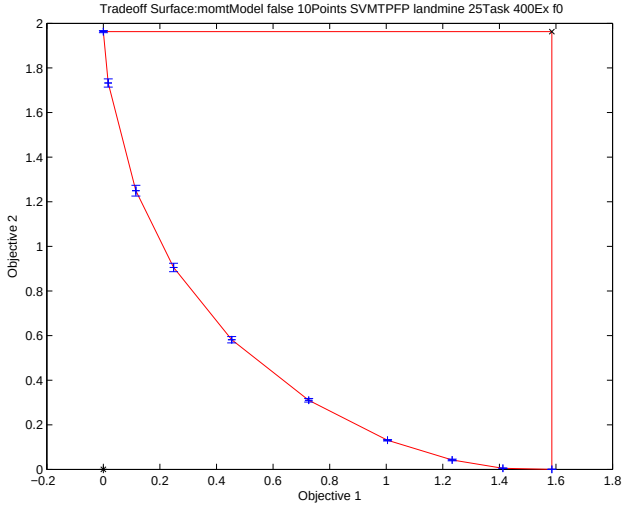
We note a few general trends in the figures. The tradeoff surface has the desired convex shape. For the artificial datasets, the distance among tradeoff surfaces increases as the variance among the tasks increases, which is to be expected. Most curves have low standard deviation, indicating that the algorithms are stable. Although the TPFP points have no apparent pattern for most algorithms, they tend to be concentrated in a small region near the ideal point for most datasets and algorithms.

Most algorithms show consistent behavior across the datasets. The regularization approach of SVM appears ill-suited to multi-objective learning because most of its accuracy v. weight curves were relatively flat. This indicates that it is insensitive to changes in objective weights. This agrees with the results presented

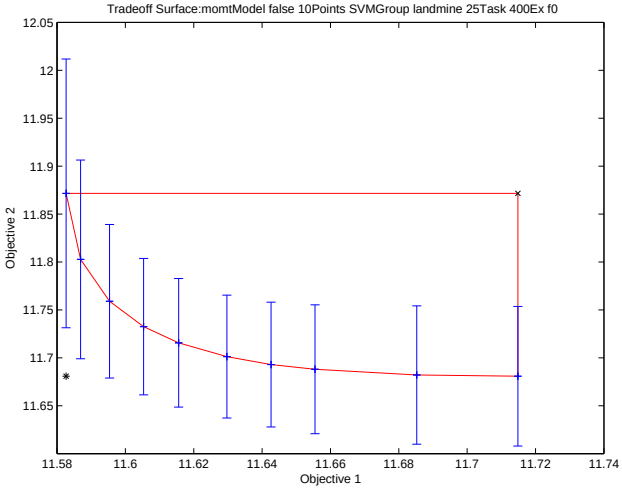


(a) DFL

(b) SVM

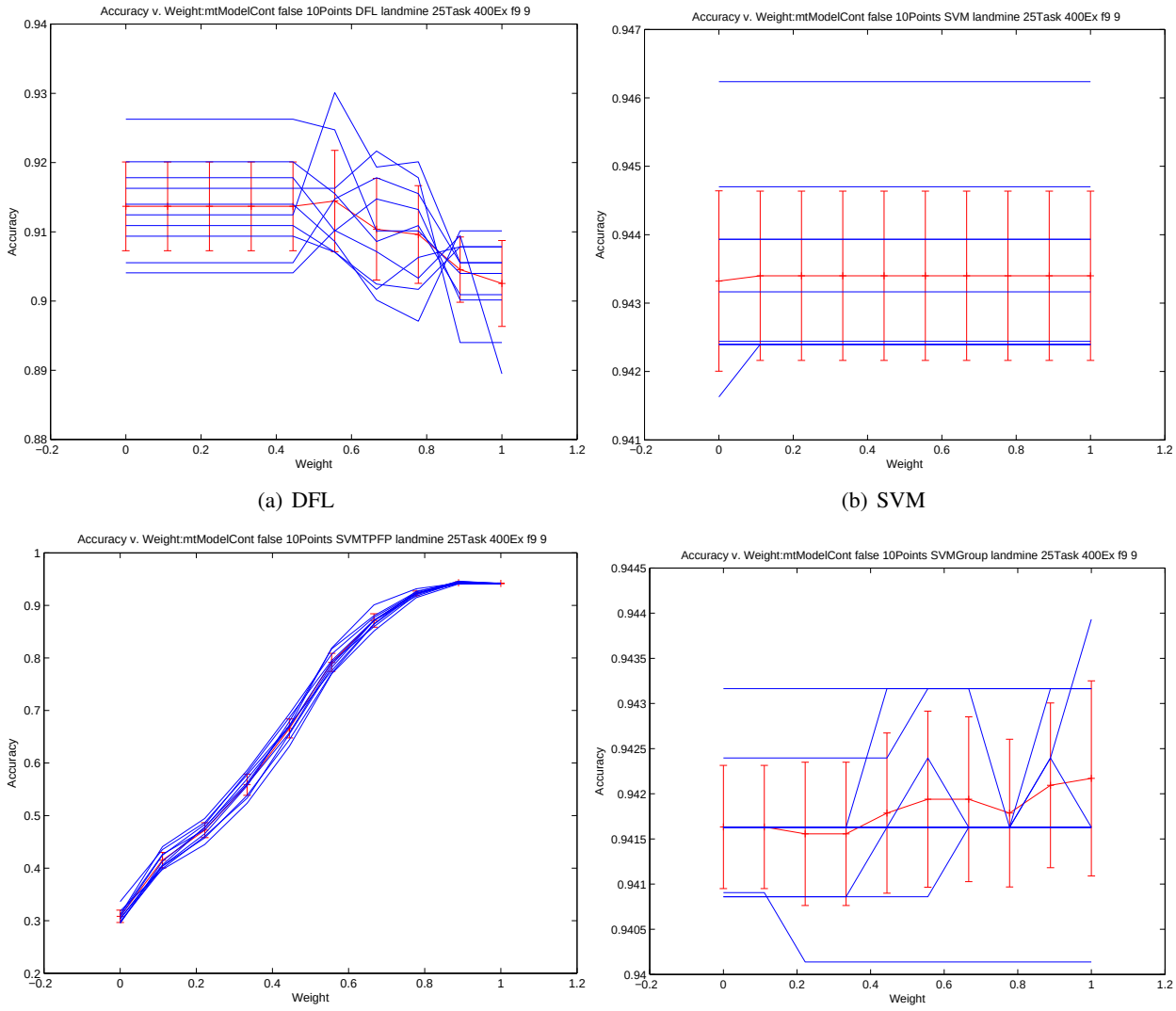


(c) SVMTPFP



(d) SVMGroup

Figure 4.13: Tradeoff surfaces on the landmine dataset for 400 examples in each dataset. The horizontal axis shows the value for the first objective; the vertical axis shows the average and standard deviation for the second objective.



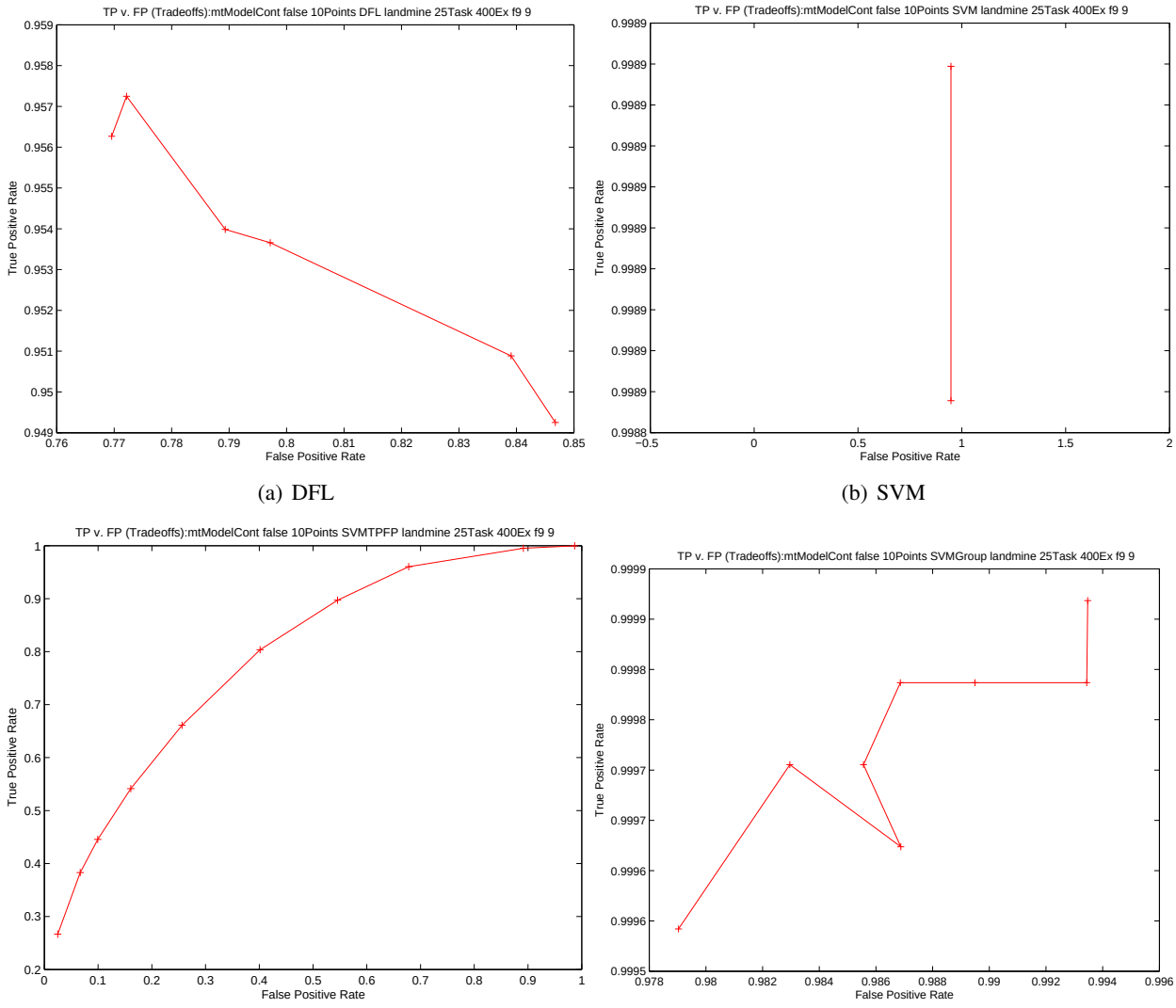
(a) DFL

(b) SVM

(c) SVMTPFP

(d) SVMGroup

Figure 4.14: Accuracy v. weights on the landmine dataset with 400 examples in each dataset. The horizontal axis shows the weight of the first objective; the vertical axis shows the average and standard deviation of the accuracy for the 10 folds.



(a) DFL

(b) SVM

(c) SVMTPFP

(d) SVMGroup

Figure 4.15: (fp, tp) pairs for the efficient hypothesis spaces for the landmine dataset with 400 examples in each dataset. The horizontal axis shows the average false positive rate; the vertical axis shows the average true positive rate, for 10 folds. A pair of points is connected by a line if the efficient points are connected along the tradeoff surface. Since each point represents a distinct efficient hypothesis space, these point sets should not be interpreted as ROC curves [34].

	DFL		SVM		SVM-Group		SVM-TPFP	
Dataset	μ	σ	μ	σ	μ	σ	μ	σ
artDatar000	68.00%	0.12	58.00%	0.08	57.00%	0.07	62.00%	0.12
artDatar010	61.20%	0.09	64.92%	0.04	65.32%	0.04	64.20%	0.04
artDatar020	68.28%	0.08	75.84%	0.05	75.92%	0.05	75.96%	0.05
artPrefS05	75.33%	0.03	93.63%	0.02	93.38%	0.02	93.13%	0.02
artPrefS10	74.63%	0.02	94.38%	0.02	93.75%	0.01	94.04%	0.02
artPrefS20	74.75%	0.02	93.71%	0.01	93.67%	0.02	92.88%	0.01
artPrefS30	75.67%	0.03	94.17%	0.02	94.17%	0.01	93.92%	0.02
landmine	91.44%	0.01	94.34%	0	94.21%	0	94.31%	0
school	49.13%	0.02	51.46%	0.03	51.80%	0.04	56.80%	0.03

Table 4.2: Average accuracy and standard deviation across all tasks. The results are shown for tasks with 100 examples.

in the original work, where most changes in error as a function of the weight ratio were extremely small [33]. Although the curves for DFL indicate that it is sensitive to changes in its weights, the poor accuracy results overall indicate that these datasets are not amenable to distance function learning. Despite its large standard deviation, the tradeoffs of SVMGroup show that the tradeoffs are not always symmetric, indicating that different clusters of tasks could lead to asymmetric loss relationships. The high standard deviation is due to random initialization in the clustering algorithm. Of all the algorithms, however, SVMTPFP shows the most promise. Its tradeoff surface is a good estimator of the expected tradeoff surface. In most of the datasets, the expected tradeoff surface shows that we can control the class prediction errors by controlling the objective weights. The TPFP plots resemble ROC curves. This allows us to leverage existing selection and analysis methods from the ROC literature for the two-class problem on multiple datasets.

Accuracy of the Efficient Set Tables 4.2 and 4.3 show the accuracy values, averaged across all tasks. The best (or one of the best) weight values were selected as the maximum testing accuracy. The results show that all SVM algorithms outperform DFL on most datasets. On average, the SVM algorithm performs best, which is not surprising because it the only one of the algorithms that minimizes training error directly. For the school dataset, SVMTPFP outperforms all others.

	DFL	SVM	SVM-Group		SVM-TPFP	
Dataset	S	D	D	S	D	S
artDatar000	1	-1	-1	0	0	0
artDatar010	0	0	0	0	0	0
artDatar020	-1	1	1	0	1	0
artPrefS05	-1	1	1	0	1	0
artPrefS10	-1	1	1	0	1	0
artPrefS20	-1	1	1	0	1	-1
artPrefS30	-1	1	1	0	1	0
landmine	-1	1	1	-1	1	0
school	0	0	0	0	1	1

Table 4.3: Significance results with respect to DFL (D) or SVM (S), based on the accuracy values in Table 4.2. A $+1$ indicates a significant improvement, -1 indicates a significant decrease, and 0 indicates that the change is insignificant.

5 Related Work

Although the problem of multi-objective multi-task learning has not been addressed directly in the literature, the importance of balancing multiple objectives is becoming increasingly obvious. Most existing multi-task learning algorithms already encounter multiple objectives, but the relation to multi-objective optimization has not been addressed in detail. Meta-learning and model selection explore ranking and weighting objectives, so they share a common purpose with the present work. Finally, multiple objectives have been addressed directly in the context of single-task learning.

5.1 Multi-Task Learning

Multi-task learning is an interesting extension to single-task learning for two main reasons. Single-task algorithms may have to be redesigned to work in the multi-task model, which is itself interesting. Secondly, it has been shown that, theoretically, multi-task learning asymptotically lowers the number of samples needed to learn each task versus single-task learning.

5.1.1 Algorithms

Multi-task algorithms learn some component that is shared across tasks, so learning each task consists of specializing from this common component. We can group algorithms for multi-task learning based on what kind of common component they learn. Although all multi-task algorithms assume that the tasks are related in some way, the exact nature of this relationship is not always explicit or well understood. Common model algorithms learn a common component of the algorithm. Common transformation algorithms learn a preprocessing operator that is applied to each task before applying a single-task learning algorithm.

Common Model

Common-model algorithms start with a single-task learner and divide the model into common and specialized parts. Neural networks were one of the earliest and most straightforward such algorithms, cross-connecting nodes for different tasks. Algorithms like Gaussian Processes can be extended by modeling the hyper-distribution, Φ, describing the tasks.

Neural networks learn a target function by training a set of weights arranged in layers. This layered structure allows great flexibility in fitting a function [3]. For multi-task learning with neural networks, the inputs from several tasks train a neural network with a common network of hidden layers but different output nodes for each task [22]. Although this allows the learner to specialize for different tasks, it treats all tasks equally. Tasks that are dissimilar could result in poor generalization performance. Instead, we can assign a node that controls whether a task contributes to the overall solution [8]. Each task has a sigmoid-type activation function that indicates when task-specific information should be used. If tasks are all very similar, the specialized information is not needed. If tasks are different, some specialization will be necessary.

In Gaussian Processes, a hypothesis for a single task can be selected from a prior distribution over the hypothesis space. In multi-task learning, one must find the task-level hyper-distribution, Φ, and then learn each task individually. Learning the individual task is easier because the size of the search space of possible models, the variance associated with the lower-level parameters, is significantly reduced [82]. The task-level

hyper-distribution can be modeled by a hierarchical Bayesian network [82], a Dirichlet process [80], or a Radial Basis Function network [53]. Radial basis function networks simplify Gaussian processes and have a similar structure to neural networks. Instead of a sigmoid activation function, the nodes are activated based on similarity (via a kernel function) to points in the training set. A common network is created for all tasks but the output nodes are different for each task [53].

Common Transformation

The transformation algorithms assume that tasks share a common feature representation. One main advantage of these algorithms is that given an appropriate transformation, existing single-task learners do not have to be redesigned to be applied to each task. Feature selection algorithms find a subset of features or weights for features. In linear transformations, examples in all tasks are projected onto a linear subspace through a matrix multiplication. In nonlinear transformations, a kernel representation is learned and applied to each task.

For tasks with many features, such as text categorization, some features are irrelevant and can be ignored, or de-emphasized, leading to better performance. The same features could be irrelevant for many different tasks. In feature selection approaches, the output of the learning algorithm is typically a different model for each task after extracting common features. In the context of support vector machines, a feature selection method can be seen as a bit-mask applied to the weight vector. Thus, the weights that correspond to irrelevant features have a value of 0. Learning a separate bit mask, however, complicates the optimization problem. To solve this problem, we can assume a prior distribution over bit masks and find the parameters for this distribution. The maximum entropy approach minimizes the entropy of the distribution given the parameters, which makes the solution tractable [44]. In feature weighting, rather than learning a 0-1 bit mask, we learn weights for the features. Instead of removing irrelevant features, they are de-emphasized. This problem can be converted into a convex problem and solved via an iterative approach [4]. The SVM algorithm is applied to the tasks assuming some initial feature weights. The weights are computed based on the weights of the SVM, and the process is repeated until convergence. In related fields such as information retrieval, feature weights are obtained interactively, through relevance feedback. The retrieval process can be improved by finding weights for groups of documents, used in similar queries [7].

Selecting features or weights for features is a special case of a linear transformation. Before applying an algorithm, each dataset is multiplied by a matrix θ, resulting in a new dataset $X'_t = X_t\theta$. The idea is that if the linear operator is "good", tasks will be easier to learn. This assumption is commonly used in conjunction with single-task learning algorithms such as singular value decomposition (SVD) and independent components analysis. These analysis methods can be applied on the concatenation of different tasks and used as a common transformation [83]. Low-rank approximations to tasks are commonly used in recommender systems. We can thus view multi-task learning as finding a projection that works for multiple tasks [25, 67, 81]. Instead of projecting the task, we can use the SVD to find a transformation of the weights for the SVM [2].

One of the strengths of the support vector machine is that it can learn non-linear functions after projecting the dataset on to a reproducing kernel Hilbert space. For single tasks, we create a kernel matrix by computing the kernel function for all pairs of examples. The learned hyperplane has one weight for each example. In multi-task learning, tasks could have different numbers of examples, so the weight vectors and kernel matrices can be very different, in each task. As a result, much work has been dedicated to redefining kernels to deal with multiple tasks. One method is to define a kernel as mapping from a pair of examples to a vector of outputs, one for each task [61, 62]. This allows us to combine multiple tasks into a single multi-valued kernel matrix. The weight vector then becomes a matrix, one for each task. Alternatively, we can use a sparsely defined kernel, a block diagonal consisting of blocks for each task [32].

5.1.2 Frameworks

Recently, a few theoretical frameworks for multi-task learning have been developed. The main result is that the average sample complexity per task decreases as the number of tasks increases. Several extensions have been proposed to tighten the bounds.

One of the earliest frameworks, due to Baxter [13], bounds the sample complexity of multi-task classification in terms of the generalized VC dimension. The generalized VC dimension is the number of tasks that are needed so that a learner learns all possible Boolean functions on all possible datasets of a fixed size. A key feature of the sample complexity is that the number of examples needed per task decreases as the number of tasks increases. This established the main theoretical motivation for multi-task learning, fewer examples are needed on individual tasks if there are many similar tasks. The framework, called bias learning, assumes that tasks are sufficiently similar such that a common hypothesis space θ contains the optimal hypothesis of each task. The bias learner learns a hypothesis space θ of a family of hypothesis spaces H from a fixed sample of tasks $\{T\}$ from a hyper-distribution, Φ. The details of the framework are discussed in Section 2.2.3. No assumptions are made on the tasks or hypothesis spaces other than "admissibility". In contrast to multi-task learning, as considered here, bias learning is concerned with generalizing to new tasks from the distribution Φ. In multi-task learning, generalization is only with respect to the fixed set of tasks.

A limitation of Baxter's bias learning framework is that it does not specify what is meant by similar tasks. Clearly, if tasks are not related, then bias learning would fail. Ben-David *et al.* considered an explicit model of relatedness, $\mathcal{F}$-relatedness, where $\mathcal{F}$ is a class of functions [14]. Tasks T_1 and T_2 are $\mathcal{F}$-related if there exists $f_1, f_2 \in \mathcal{F}$ such that $T_2 = f_1(T_1)$ and $T_1 = f_2(T_2)$, where equality is assumed. Under this framework, the sample complexity of each individual task was shown to decrease as the number of tasks increases. Thus, if tasks are related, we can learn on one task and immediately transfer that knowledge to another task by means of the relatedness function. The theoretical results were demonstrated on a hypothetical algorithm and relied on Baxter's generalized VC dimension.

In order to tighten the bounds, recent work has investigated data-dependent bounds, extending results from single-task learning to multi-task learning. In both previous frameworks, the bounds depend on a generalized VC dimension, which depends only on the algorithm, not the dataset. It can be difficult to compute even in the single-task case. Data-dependent bounds can be shown to improve on data-independent bounds. Two such bounds depend on the β-stability [19] and the Rademacher complexity [12]. Section 2.2 discusses the Rademacher complexity in detail. The β-stability bounds the average training error of a learning algorithm. An algorithm is said to be β-stable if the average leave-one-out loss on a dataset is at most $\beta > 0$. Intuitively, the larger β, the less stable the algorithm is due to changes in the training set. In multi-task learning, Maurer extended the notion of β-stability to multiple tasks, resulting in two stability parameters β' and β. A multi-task learner is β'-stable if the difference between the average leave-one-task-out loss and the full loss is at most β'. One-sided bounds were then shown to depend on these constants. Although the context was on meta-learning rather than multi-task learning, the results are related.

5.2 Meta-Learning

Unlike multi-task learning, in which a common hypothesis space or transformation is shared across tasks, meta-learning algorithms select a learning algorithm to apply to a new task. What is transferred between tasks is the mapping between tasks and algorithms. Meta-learning research has three main branches. First, features are extracted from a trained algorithm, such as accuracy. Second, features are extracted from a dataset before any algorithm is applied. Finally, the learner selection problem is viewed as a retrieval process, methods from information retrieval are employed to select algorithms.

5.2.1 Performance

The performance of several algorithms across different tasks provides useful information for selecting algorithms, called the base learners. In landmarking, simple (landmark) learners are trained on each task [15]. The average cross-validation accuracy of each learner forms a feature vector for the task. Associated with each dataset is the label of the base learner that has the highest average performance on the dataset, or a tie label indicating that there is no significant winner. Several standard classification algorithms can be applied to this meta-dataset to determine the appropriate learner for a new task [38]. The advantage of this method is that the landmark learners have low computational complexity and are suited to different tasks. Thus, we can apply them to new tasks without having to apply the more costly base learners. One of the limitations of this feature-vector approach is that there are no features related to tasks. Specifically, the performance of the learner does not take into account the sample size. A dataset that has high error for n examples on a simple learner could have much lower error for more examples, say $2n$. The landmarking system ignores this fact. A recent extension is to associate a learning curve with each algorithm [51]. A learning curve shows average performance after training on a sample with n examples, where n increases from some minimum number, say 10, to the number of examples in the dataset. The learning curve can be subdivided into ranges, such as 10% to 100% of the training set. This forms a feature vector that can be used, like landmarking, to determine which algorithm should be applied to the task.

5.2.2 Task Similarity

A common belief in meta-learning, and much of case-based reasoning, is that similar problems have similar solutions. In the case of meta-learning, this means that similar tasks can be learned by the same base learner. Defining a distance function over tasks is the subject of current research.

One method is to extract features from the tasks. Once the features are extracted, the distance between a task is defined as the distance between the feature vectors. Examples include statistical features such as the number of examples, number of attributes, and average of each features [75]. Information theory provides several additional features, such as the class-conditional entropy of the features [20].

A more direct approach is to define a distance function directly between tasks. Since tasks are a probability distribution, many techniques from information theory can be used. In terms of Kolmogorov complexity the lower bound of the distance between two objects is their compression distance. In the compression distance, we compress task T_1 using the codewords from task T_2 and *vice versa* [52]. The compression ratio between these compressed sizes tells us how much the two tasks have in common. The compression distance gives a lower-bound on the transfer between tasks in multi-task learning, and can be used as a distance function for meta-learning [46]. The relative entropy measure (also known as Kullback-Leibler divergence) measures the entropy of encoding task T_1 assuming that its distribution is given by T_2 [24]. Instead of measuring the entropy directly on the task, we can measure it instead on the output of a function trained on the dataset [9].

5.2.3 Selection Methods

After creating a meta-feature set to represent a trained learner and a task, the next challenge is to select the best base learner for a new task. Recognizing that the recommended learner must satisfy several objectives, much work is devoted to ranking learners. A user often has some knowledge of which meta-features are more important for ranking algorithms, and the user can customize a ranking function by providing weights for the meta-features. For a new task, learners are ranked according to their distance under the user-defined weights [66]. Often, however, the user cannot simply specify weights. Instead, users have rank profiles that specify a preference order over algorithms [16]. Weights are associated with the profile, and when a new task arrives, the weights from all profiles create several candidate distance functions. The learners are

matched based on the rank correlation between the learner's rank and the user's preference. Thus, the user does not have to specify a profile, only evaluate the results across profiles. Rather than a retrieval approach, one can predict the rank of an algorithm on a new dataset [20]. A predefined distance function returns learners within some pre-defined distance to the query task. The rank is estimated by the average of the ranks of each learner on similar datasets. The top k learners are returned.

5.3 Model Selection

In model selection we seek an optimal hypothesis space or transformation for a single task [47]. Although it is not typically considered a multi-task learning method, both seek an optimal hypothesis space. Unlike multi-task learning, the hypothesis spaces are assumed to have a structural relationship.

Model selection is defined within the principle of structural risk minimization [76]. As the structural complexity of the learner increases, the training error decreases but the testing error can increase. We seek to balance the complexity of the learner against the training error, with the hope that this will not over-fit the dataset. Thus, there is an inherent risk associated with the structure of the learner, independent of the dataset [71]. As an example, consider separating examples in two classes by a hyperplane. If the training dataset is separable, we can always find such a hyperplane. In fact, we can often find several hyperplanes that yield the same error rate. The structural risk minimization principle requires us to consider the simplest such hyperplane, which has been shown to be inversely proportional to the norm of the weights [76]. Thus, we seek to first minimize empirical error and then to minimize the norm of the weight vector. There is a hierarchical relationship between the hypothesis spaces. Specifically, the weights satisfying the condition $\|w\| \leq C$ also satisfy $\|w\| \leq C'$ where $0 \leq C \leq C'$. This means that if we find the minimum-norm weight vector, we can be confident that any larger-norm weight vector does not fit the training data better.

Relying on this structural relationship between hypothesis spaces, model selection algorithms have been designed for many different learning algorithms. Because of the established theory for support vector machines, much work in model selection has been applied to find the appropriate kernel function for a task. A meta-learning approach finds the kernel width for new tasks using results from training tasks [72]. We can also find the appropriate degree for the polynomial kernel with an incremental approach [70]. A distance metric over kernel functions is defined over the output of the learned function. As the degree of the kernel increases, the decision boundary becomes increasingly non-linear, causing the distance function to no longer satisfy the triangle inequality. When this occurs, the degree is too high. The selected model is the largest degree that still satisfies the triangle inequality for lower degrees.

A key difficulty in implementing model selection algorithms is quantifying the relationship between hypothesis spaces. Since increasing complexity tends to decrease training error, we cannot use training error as a stopping criterion for model selection. Instead, we would like to be able to estimate the testing error for our hypothesis space. The Rademacher complexity makes it especially easy to estimate the testing error for medium datasets, requiring only the ability to train the learner on a dataset with different class labels. For small sets of hypothesis spaces, we can compute the Rademacher complexity on all of them and take the minimum [49]. More specialized estimation methods can be used to obtain tighter bounds [10]. The Kullback-Leibler divergence can bound expected loss and is used for model selection [39, 59].

5.4 Parameter Selection

In contrast to the assumed relationship in model selection, parameter selection algorithms are often required to search large parameter spaces Recent work employs regression algorithms to predict performance on new parameters, thus saving evaluation time. Multi-task feature selection in SVMs learns a bit-mask applied to the weights that is useful for multiple related learning tasks [44]. Regression modeling has been used to

predict the performance of parameters for a single dataset. An online Gaussian processes regression model is trained on the error rate of the SVM classifier for a sample of parameters [37]. The regression model is augmented by evaluating new parameters based on their expected improvement to the error rate. In an interactive model of multi-task learning, all tasks are not available at startup. Under these conditions, an incremental model of the search space is stored, mapping paths in the search space that led to high objective values [6]. As a new task arrives, the model is updated and allows faster convergence to a good solution.

5.5 Multi-Objective Learning

Single-task learners have been extended to handle multiple objectives by learning vector-valued functions, or learning the tradeoff surface.

5.5.1 Vector-Valued Loss

Multi-class learning algorithms, in contrast to single-class learners, assume an example belongs to multiple classes at the same time. Instead of predicting a single class value, these algorithms predict a vector of class values. We can model a multi-class problem as vector-valued optimization, minimizing the empirical loss across all classes. This is shown to lead to a valid learning algorithm, assuming that there is a hypothesis that minimizes loss for all classes [74]. This is in contrast to the multi-objective approach in which we explicitly assume that there is no single a hypothesis that minimizes loss for all classes. For a dataset with just one class label but multiple classes, ROC curves are extended to multi-dimensional surfaces. Here we are interested in maximizing the true positive rate in each class. The ROC surface is the tradeoff surface of the performance of each class with respect to decision thresholds [31]. When the loss function is single valued but depends on multiple variables, such as the F-measure or the Precision, it can be difficult to formulate this as a single objective function. Instead, we can decompose our multi-valued loss function into one vector-valued loss function for different class labels. We can then use vector-valued optimization or scalarization methods to perform the optimization [45].

5.5.2 Pareto Optimal

It may not be possible to learn all classes equally well; thus, a compromise is necessary. Pareto-optimal single-task learners learn hypothesis that is best for each class individually. Even the standard SVM can be viewed as a multi-objective optimization problem, minimizing the norm of the weights and the empirical error rate [17]. In binary classification, our ultimate goal is to minimize the probability of making an error in each class. Under the assumption that each class belongs to a single distribution, we can find a hyperplane that is equally distant, with respect to the Mahalanobis distance, to each class. This optimal hyperplane is Pareto optimal in the sense that it minimizes the expected error for both classes [48].

5.5.3 Multi-Objective Optimization

The proposed work uses existing multi-objective optimization algorithms to do multi-task learning. The goal of a multi-objective optimization algorithm is to determine the tradeoff surface within a set of constraints [29]. More sophisticated search methods have been proposed for larger problems, such as evolutionary computing [26, 27, 54, 84]. In these methods, convergence is defined by the distance to the true tradeoff surface. This is quite different from the present work. Here, the true tradeoff surface, which we call the expected tradeoff surface, is only available when all examples are known. We do not typically know the expected tradeoff surface. The optimization algorithms only operate on the training set, a finite sample. The true tradeoff surface of interest to optimization algorithms is the empirical tradeoff surface. The main

concern of optimization algorithms, in our domain, is whether the output of the algorithm converges to the empirical tradeoff surface. Our work concerns a different problem. We are concerned with whether the empirical tradeoff surface, output by some optimization algorithm, converges to the expected tradeoff surface, which is not available to the optimization algorithm. A related problem is whether, by applying the optimization algorithm to larger and larger samples, we can expect to converge to the expected tradeoff surface.

5.6 Evaluation Methods

When two multi-objective learning algorithms optimize the same objectives, their tradeoff surfaces can be compared. Comparing tradeoff surfaces is often done in the field multi-objective genetic algorithms. Two evaluation methods are popular: comparing tradeoff surfaces and evaluating the quality of a single tradeoff surface.

In evaluating a single tradeoff surface, we would like to reduce the entire tradeoff surface to a scalar value. A simple method, frequently used in machine learning for ROC analysis, calculates the area under the tradeoff surface [34]. This gives us a scalar value with which to compare algorithms. Computing the area for multi-dimensional surfaces can be expensive, however [42]. If the optimal tradeoff surface is available, methods such as the proportion of points on the tradeoff surface are popular [27].

6 Cougar^2 Software Library

The purpose of the Cougar^2 software library is to simplify the transition from algorithms on paper to efficient and reusable software. Most research in machine learning comes to a point at which ideas need to become executable software, in which there are typically two paths to follow. Packages like Matlab are designed to simplify the process of transforming an algorithm from an equation to code with as little programming as possible, often sacrificing speed and usability [56]. Packages like WEKA and Yale emphasize usability to the extreme, catering to the the casual user with graphical interfaces and plug-ins, requiring the researcher to design graphical user interfaces instead of algorithms [64, 79].

6.1 Design Goals

What distinguishes Cougar^2 from other packages is a primary design goal of being a programming library rather than a graphical interface. We have learned several lessons from other software and designed the software to make common operations easy. At the core is a set of programming interfaces from which a variety of algorithms can be created and reused. As a library, we have three goals for the code which are (in order):

1. Easy to add new types of algorithms.
2. Follow test-first development.
3. Write self-documenting code.

Some progress toward these goals has been made but much remains to be done. For this reason, the project has been released to the open source community, allowing anyone to join and potentially contribute.

6.1.1 Lessons Learned

The library was designed to improve upon existing libraries. Several important lessons were learned from existing packages and applied to Cougar^2.

Many packages have designed their class hierarchy to model the conceptual differences in different learning algorithms. For example, supervised and unsupervised learning algorithms are quite different conceptually. In other packages, these are distinct and typically incompatible class hierarchies. Instead of modeling the conceptual differences in software, we model differences in how the algorithms are used. Both clustering and classification algorithms take the same input and typically have the same output. The difference is that classifiers read the class labels whereas clustering algorithms write them. In analyzing the results, it is important to read the class labels from a clustering algorithm, and both models can be used to label incoming examples. In our design, only the model needs to be different for these different types of algorithms. Thus, different types of algorithms do not have to be part of an all-encompassing class hierarchy. Instead, we prefer an all-encompassing design philosophy. Algorithms have distinct phases with well-defined interactions. Regardless of the algorithm, these phases do not change even if the parameters or functionality changes.

For researchers developing new algorithms, interacting with graphical user interfaces can quickly become tedious and inefficient. The two most popular packages, WEKA and Yale, have expended significant effort

in designing graphical interfaces [64, 79]. Although useful for the casual user, researchers can typically write code to do most of the common tasks faster than they can use a GUI each time. In creating intuitive user interfaces, much has been sacrificed in terms of ease of development and type safety. In WEKA, there are two distinct configuration interfaces that each class must support, standard Java properties and a command-line interface. In order to support properties, an algorithm class must have a no-argument constructor. This makes it impossible to do checking in the constructor, forcing the developer to remember to call a check method, if one exists. In WEKA, no explicit check method exists, so configuration errors are caught only at runtime. In our design, the configuration is part of a separate class called the Factory, which stores the properties and calls the constructor with all the arguments. Instead of standard Java properties, a Yale algorithm must fit into a hierarchy of operators. The operator paradigm removes much of the type checking available to Java, preferring to interact via String-based properties. A new algorithm must implement the Operator methods which add to the standard properties. This is another example of presentation-code finding its way into the design of algorithms. Although our experiment system has an operator-like syntax, it is completely isolated from the rest of the API. So, an algorithm does not need to know if it will be used in an experiment or a GUI. No special handling is necessary.

Recently a new software development practice has emerged, agile development [55]. The strategy emphasizes a firm commitment to a few design principles. Although many existing packages were developed before the popularity of agile development, our project was designed with agile practices in mind. The most applied principle is the Dependency Inversion Principle, which states that classes should depend only on interfaces [55]. Next, we apply the Interface Segregation Principle, so that clients only depend on the functionality in interfaces they use [55]. The main benefit of these principles is that they facilitate unit testing by isolating a class from changes in the implementation of other classes. When implementing a new class, any errors that occur should be the result of the current class and not some other class over which we have no control. Following this principle requires that the core API contain only interfaces. This level of abstraction is not possible in WEKA [79]. The most used class in WEKA is probably the dataset class, Instances. This is a concrete class with approximately 2,000 lines and almost 100 methods. Since it is concrete and has several public constructors, unit testing with a dataset is extremely difficult. It usually requires creating a file or a method that creates a dataset with a specific property. Simulating errors in data access is especially difficult if not impossible. In contrast, our Dataset is an interface and several utilities exist to facilitate creating mock objects for testing.

An important feature enabled by unit testing and agile development is white-box development. In our library, this refers to designing class so that most, if not all, methods are public, short, and do not directly depend on member variables. In C++, most of these methods would be declared `const`, indicating that they do not directly alter members of the class. These methods are easy to test because we can create mock objects in the test and pass them to the method without worrying about side effects. They are also easy to use within the class because we can pass member variables as arguments. They also reduce coupling between concrete classes because most methods only depend on interfaces.

6.1.2 Common Operations

The design of the library is the result of our experience working with other libraries, where common operations need to be easy to write and run. The three most common operations during typical development and experiments are: configuring a learner, traversing through a dataset, and accessing common utility methods.

When creating an experiment and when developing an algorithm, a common operation is to configure a learner. There are two goals in this operation. Obviously, we want to be able to instantiate a learner and set some properties. One approach is to use a constructor with all arguments. This is problematic for two reasons. First, it requires that we access the concrete implementation of the learner which couples the caller to the implementation of one method in our learner's class. Second, the caller has to remember the order

of the parameters and what they mean. For complex objects it can be difficult, for example, to remember the difference between the third and fourth integer parameter. This is made worse if the constructor ever changes. The second goal of configuring a learner is to store the configuration for later analysis. With a constructor, the code itself acts as a record, unless the code is changed. Our design places the creation of a learner into a separate Factory class. The Factory contains named properties, so that one does not have to remember the order of the syntax and can set the properties in any order. The factory, being a separate object, can be stored and used as a prototype, allowing us to examine the configuration or to create identical objects in the future.

In developing a learning algorithm, one of the most common operations is to traverse a dataset. Typically this means writing a `for` loop and applying some operation to each example. In order to access a value from the dataset in WEKA, we have to acquire an Instance object for the example and retrieve the value from the Instance. This introduces a performance overhead, compared to accessing data in an array. It also forces a single view of the data, row-major order. With an array in column-major order, we can load all values for some attribute and traverse most datasets in contiguous memory. Our Dataset API does not constrain the developer toward a particular organization of data. All that we require is random access. Another desired feature for a library is to facilitate using the `for-each` construct, now available in Java 1.5. A method adopted in other APIs is to iterate over the values, example objects, but we often want to iterate over values and then record the index, such as computing the index of the item with the maximum value. In C++, this problem was solved with the `distance` method, but we know of no similar method in Java. As a result, our Datasets return iterators over indices, so that we can avoid writing simple loops, which are prone to errors.

Much of the functionality of existing algorithms can be extracted into static methods that can be reused, commonly referred to as utility methods. The obvious benefit to these methods is that they can be reused without side effects. The disadvantage is that static methods make it extremely difficult to do unit testing and it couples a method to a concrete implementation. For example, suppose that as part of a learning algorithm, we need to split a dataset into an array of datasets based on their class labels. This could be done by calling a utility method, but if the Dataset is a mock object, we have to mock all the calls necessary to implement the utility method. This can be extremely problematic because we now have to mock the behavior of the utility method in addition to our originally tested method. Our solution to this problem is to design utility methods as part of a utility interface. We can mock the utility class, so that now we only have to test the class we intended to test. The drawback to this approach is that methods are no longer static but require an object. In addition we have to define an interface method for each utility method we will apply. This is often a source of frustration for new developers.

6.2 Core API

The library is separated into two core models. Datasets deal with how to read and write data. Learners process the data and return reusable results.

There are a few principles that guide the design of the modules. The API consists mainly of interfaces, which are designed to be used with test-first development, which is the primary development strategy. We deliberately chose a short and wide interface hierarchy, but it is structured consistently. We follow a white-box development methodology, meaning that we prefer that all methods in a class be public with as much information passed through parameters as is practical. The main reason for this is that the interfaces are the main public API, but if extra functionality is already implemented in a class it should be reused as much as possible. Eventually, the functionality will be refactored into a separate utility class if it is frequently used. This development strategy reduces side effects and promotes reusability of the methods.

Algorithm 4 The NumericDataset interface, showing operations specific for dealing with real-valued data.

```
public interface NumericDataset extends Dataset {
  public NumericAttribute getAttribute(int iAtt);
  public double get(int iRow, int iAtt);
  public void set(int iRow, iCol, double value);
}
```

Algorithm 5 The NominalDataset interface, showing operations specific for dealing with integer-valued data.

```
public interface NominalDataset extends Dataset {
  public NominalAttribute getAttribute(int iAtt);
  public int get(int iRow, int iAtt);
  public void set(int iRow, iCol, int value);
}
```

6.2.1 Datasets

Although the purpose of a Dataset object is to retrieve and possibly store data during the learning process, there are many additional requirements. From an efficiency perspective, the input-output operations should be fast, suggesting arrays or databases. From the development perspective, reading from a dataset is probably the most common programming task in machine learning. Consequently, it should be efficient to *write* the code as well as to run it. From an algorithmic perspective, algorithms make certain assumptions about the dataset, which should be reflected in the design of the code. Most of the algorithms we use assume that the data are all available at run-time, running in batch mode. For example, a decision tree like ID3 assumes that the data consists of discrete (nominal) values [28]. It does not care how the data are represented in memory as long as it can find the number of discrete values for an attribute and tell if two are the same. In contrast, a support vector machine only functions if the data are continuous (numeric) [76]. Here, there is no use in counting the number of discrete values, but knowing the minimum and maximum values might be useful. Consequently, an algorithm should only depend on the features of a dataset that it needs.

Our design for a dataset is to create several different and incompatible dataset interface hierarchies, illustrated in Algorithms 4 and 5. By different, we mean that each interface specifies how an algorithm perceives the data. A NumericDataset returns all values as a `double` value, and a NominalDataset returns all values as an `int`. By incompatible, we mean only that one cannot assign a NumericDataset to a NominalDataset variable. The interfaces are, syntactically, almost exactly the same, so that one can easily transform a learner from one interface to another without having to rewrite much of the code.

The interface for a Dataset provides a few operations that are common to all subtypes, as shown in Figure 6. The first are random access operations like get and set a value. A dataset is a two-dimensional table partitioned into groups, as illustrated in Table 6.1. First are the "real" data, the values that a learner would use directly for learning. Next are the meta-data, extra information added by users. The examples of meta-data could contain a class label or a task label. Meta-data stay with the dataset but may not be used in the learner. The next group of methods are iterator operations that facilitate using the new for-loop construct in Java 1.5. Finally, there is an allocate method that returns an empty dataset object, which allows a Dataset to be used as a prototype for new datasets.

Several operations and features present in other packages are noticeably absent from this interface. There are no example-level objects. In most existing packages, a Dataset consists of several example objects

0 Num	1 Num	2 Num	0 Nom	1 Nom
1.5	2.0	0.7	"Positive"	0
0.3	6.7	5.4	"Negative"	0
8.7	6.9	6.1	"Negative"	1
7.8	9.4	3.7	"Positive"	1

Table 6.1: Dataset structure, any arbitrary object can appear as an entry. Meta-data and real-data are indexed separately, as indicated by the columns.

Algorithm 6 The Dataset interface, showing operations common to all dataset objects. Examples are rows of a matrix and entries are objects. Subtypes can make assumptions about the types stored in a Dataset.

```
public interface Dataset {
  public int size();
  public int getNumAttributes();
  public Attribute getAttribute(int iAtt);
  public Attribute getMetaAttribute(int iAtt);

  //Random Access
  public Object getObject(int iRow, int iAtt);
  public void setObject(int iRow, iCol, Object value);
  public Object getMeta(int iRow, int iMetaAtt);
  public void setMeta(int iRow, int iMetaAtt, Object value);

  //Iterator Access
  public Iterable<Integer> examples();
  public Iterable<Integer> attributes();
  public Dataset allocate();
}
```

(called the Instance object in WEKA [79]). The idea is in the spirit of object-oriented design. In practice, however, these objects require storage overhead that is often impractical for large datasets. The objects can be located at arbitrary locations in memory which can slow down some algorithms. For different storage methods, such as database access or file-based access, requiring example objects increases the burden on the developer. Our interface represents each example as an integer index into the dataset. This gives us considerable freedom in storing the data, as a contiguous column-major array in memory or in a database. The second obvious omission is indexing the attributes by a String name. Usually, attribute names are only needed when looking at the data values in a spreadsheet or a file. They should never be used in developing algorithms, so we do not include them. The file-based representation of the dataset can store attribute names for manual inspection in separate specification files. Another feature, common in WEKA [79], is to add utility methods inside the interface. This violates good design practices making the interface too large and making classes dependent on features they may never use. Most of this functionality has been moved to separate specialized utility packages.

6.2.2 Learners

During our development of learning algorithms, we find that a learner has three distinct phases: configuration, learning, and result. In the configuration phase, a user sets various parameters, which may or may not result in a valid configuration. The learning phase takes the data, creates the learning hypothesis and returns it. The result phase consists of further analysis of the learned hypotheses, often long after the learning phase is complete. Our design makes a learner a component consisting of one class for each of these phases: the Factory, Learner, and Model.

To configure an algorithm, one must interact with the Factory. The Factory object for a learner must implement a `create` method, which checks the parameters and returns a valid learning object. The Factory optionally contains extra properties. For example, a support vector machine contains a property for the kernel function. Once the configuration is complete, the create method returns a learning object. The create method is provided the dataset on which to perform the learning. Its purpose is to (1) check the consistency of the parameters and (2) check whether the parameters are consistent with the provided dataset. Upon completion, the create method makes the following promise: the resulting object is properly configured and will execute on the dataset (barring problems with the dataset). If this promise cannot be met, the create method throws an exception. The reason for this strong promise is to avoid creating learners that fail at runtime for preventable reasons. The factory can be run repeatedly to test the configuration before running the learner.

The learning class consists of a single method, `call`, which takes no arguments and returns the learned hypothesis. All arguments are provided through the constructor of the learner, by way of the factory. By delegating configuration of the object to the constructor, Java lets us prevent creation of a mis-configured object. This promotes a safer design.

The result class (the Model) is designed to be stored offline for later analysis. It must contain whatever is necessary to evaluate the algorithm, the learned hypothesis and any debugging or performance information. This can be extremely simple in the case of a support vector machine, a vector of weights, or complex for a nearest-neighbor algorithm, an entire dataset. Several different model interfaces are available to support different analysis methods. For example, a classifier can just output class labels for a new example or it can provide a score. Extra information can be stored in the model after testing.

Dependencies

The main motivation for splitting a learner into distinct classes is to reduce dependencies in the code. Placing the three phases of a learner into a single class, as done is WEKA, has several limitations [79]. During development and experiments, the learning code changes the most. A goal of our design is to lessen the impact of these changes on the configuration and result code.

If the learning code changes while the experiment is running, the internal structure of the learner could be different by the end of the experiment. Even if the output of the learner did not change the resulting object may not be readable because of Java's serialization. By separating the model from the learning code, we isolate the model from changes in the learner. This also allows us to reuse the model, having multiple learning algorithms learn the same type of model.

Another important dependency is the configuration. Before running an experiment, we have to set several parameters. For large experiments, we need to keep track of which parameters led to which model. We can do this by storing a copy of the configuration object as part of the experiment. By coupling the configuration code with the learner, data structures used only during learning could be stored in the configuration. In addition to wasting storage space, this can leave the configuration unreadable if the learner's code changes. At worst, the learner could be improperly configured if important member variables are introduced after serialization. Because Java serialization does not call the constructor, these variables will be initialized

Algorithm 7 Majority classifier factory, showing one user-configured property and the create method.

```
public class MajorityClassifierFactory implements LearnerFactory<Dataset>
{
  public MajorityClassifier create(Dataset data, int iClassAtt) throws
ClassifierDatasetException
  {
    return new MajorityClassifier(data, iClassAtt,
    getWriteExtraInfo());
  }
  public boolean getWriteExtraInfo() {/*...*/}
  public void setWriteExtraInfo(boolean x) {/*...*/};
}
```

to `null` values. The learner would be readable but may mysteriously crash at runtime. Separating the configuration into a factory class isolates the configuration from the learner. It also promotes safety because the constructor will be called whenever a learner is created even when the factory is read from the disk. Thus, the learner is never serialized, only the factory and model are serialized.

The result of splitting the learner into classes is that the only way to run a learner on a new dataset is to create a new learner through the factory. This is desirable because it saves the developer from making the learner reset its state for each call. Instead, the developer can assume that the learner will only be called once on the dataset. Repeated calls for the same dataset are acceptable, however.

Example

In this example, we show how to implement a simple classifier which returns the majority class of the training set. We will create each of the three classes: a factory, a learner, and a model.

In the factory, this learner exposes just one user-defined property, whether to print out extra information about the examples. The create method checks a few things about the dataset. In the implementation, however, these checks are located in the constructor of the learner class. First, the dataset must contain at least two classes because we are doing classification. Second, there must be at least one example. The listing for the Factory is shown in Algorithm 7 and the constructor is shown in Algorithm 8. Note that the create method accesses the dataset but returns a response quickly, so the configuration fails fast. We are casting the class attribute to a nominal attribute, which will throw an exception if our dataset is not a classification dataset. This algorithm operates on generic datasets, so we only use generic datasets.

We define the learner class in Algorithm 9. Despite its simplicity, it illustrates most of the key features of our learners. The constructor performs all the checking so that the only way to obtain a valid learner object is to correctly configure it. The `call` method is the only method in the Executable interface. In this method, we simply find the most frequent class by traversing the entire dataset. The code is extremely compact to write and the iterator approach makes it especially easy. There is a performance hit associated with the cast from a meta-value (Object) to class label (Integer) and finally unboxing to int. In most implementations, however, reading meta-values is much less common that reading real data, which is directly accessed with primitive types.

In order to evaluate the hypothesis of the new learner, we apply it to some dataset, as shown in Algorithm 10. A model must write the class label at the specified meta-index in the testing dataset. This index is typically not the same as the class index used for training, but the dataset could be the same object. This

Algorithm 8 Constructor for majority classifier learner.

```
public MajorityClassifier(Dataset data, int iClassAtt, boolean info)
throws ClassifierDatasetException {
  if (data.size() == 0)
    throw new ClassifierDatasetException();
  NominalAttribute attClass = (NominalAttribute) data.getMetaAttribute(iClassAtt);
  if (attClass.getNumKeys() < 2)
    throw new ClassifierDatasetException();
  /*...*/
}
```

Algorithm 9 Majority classifier learner, simply counting the number of examples in each class and creating the model for the most frequent class. The function `maxIndex` is not shown in the implementation above.

```
public class MajorityClassifier implements Callable<MajorityClassifierModel>
{
/*...*/
  public MajorityClassifierModel public call() {
    int[] classFrequency = new int[_nClass];
    for (int i :  _data.examples())
      ++classFrequency[(int) _data.getMeta(i, _iClassAtt)];
    int mostFreqClass = maxIndex(classFrequency);
    if (_writeExtraInfo)
      writeExtraInfo();
    return new MajorityClassifierModel(mostFreqClass);
  }
}
```

Algorithm 10 Majority classifier model, setting the class label for each testing example to the most frequent class.

```
public class MajorityClassifierModel implements PredictionModel<Dataset>
{
  private final int _freqClass;
  public MajorityClassifier(int freqClass) {
    _freqClass = freqClass;
  }
  public Dataset apply(Dataset test, int iTestClass) {
    for (int i :  _test.examples())
      test.setMeta(i, iTestClass, _freqClass);
    return test;
  }
}
```

facilitates evaluating the accuracy in that now we only have to compare class labels.

Both the learner and model classes follow a common implementation pattern. First, the constructor is heavy, meaning that it performs a significant amount of checking. This promotes the creation of valid objects. Second, the member variables are declared to be final, meaning that their values cannot be changed. This helps reduce coding mistakes in assigning values inappropriately. Since all the configuration is done in the factory, there is no need to for the user to change any properties of the learner or model once they are obtained.

6.3 Experiments

An important task in this dissertation is creating and running experiments. The experiment package for Cougar[2] has been designed to meet the following goals:

- Easy to configure and adapt to changes in code or configuration.
- Runs unattended and can be resumed if interrupted.
- Results are isolated from changes in the learner and can be stored for later analysis.

Most of these goals comes from the design group's collective experience in running and analyzing experiments. In existing packages, small experiments are easy to create and sometimes easy to analyze. For large experiments, however, managing changes to code and configuration and detecting errors can quickly become impractical and tedious. To motivate these design goals, consider the following example.

(6.1) **Example**: Suppose that we want to compare our new algorithm, from Section 6.2.2, against 3 existing algorithms. The typical experiment in machine learning is to run 5 runs of 5-fold cross validation for each dataset. Our new algorithm has 2 parameter combinations, and we want to compare each of them. We can use a benchmark set of 10 datasets. In total, then we need to create and train $5 \times 5 \times (3 + 2) \times 10 = 1,250$ models.

Configuring the experiment is perhaps the most time-consuming (for people) aspect of running an experiment. In the example, we have $1,250$ models to build. Manually creating each model is obviously impractical. We want to specify the configuration: datasets, runs, folds, and algorithms as simply as possible. We will be developing the code while the experiment is running, checking the results, and fixing errors. If some of the results are bad, we would stop the experiment and run it again with the new code. We do not want to have to re-configure the experiment each time we make a small change. Often, we will see that datasets need to be changed or we suddenly want to test a new parameter combination. We want the experiment to redo only the changed parts without re-running the parts that have not changed. Finally, if there is a configuration error in the experiment, due to bad parameters, we want to know about it before the experiment starts running.

After the experiment is configured, we will run it unattended. In the example, $1,250$ models could take a long time to run. Suppose that each model can be built in around 3 minutes. The entire experiment takes more than one hour to complete. Clearly, we do not want to manually supervise the experiment. Suppose there is a runtime error in the $1,246$th model to execute. If the error is not serious, we would like to skip this model and continue with the experiment even if it will be incomplete. If the error is serious, such as an out-of-memory error, we want to be able to fix the error and resume the experiment where it stopped. Once the experiment completes or crashes, we want to see a log of actions to follow the progress or the track down the source of errors.

If the experiment takes a long time to complete, we do not want to wait until all steps are complete before beginning the analysis. The analysis code can be developed while the experiment is running, so intermediate results should be available. As the configuration is changed, some results may be obsolete, superseded by more recent results. These obsolete results should be removed or at least ignored in the analysis. We want to isolate our models from changes in the learner and factory so that even if these classes change, we can continue working on the analysis code with the experiment results we have at the moment.

6.3.1 Experiment Graph

The experiment is represented as a graph of dependencies. Most automated build systems, like Make or Ant [36], utilize graphs for tracking dependencies. There are two key benefits of a graph structure. First, parts of the graph can be isolated from the rest so that the effect of updates is controlled. Second, explicit dependencies are declared so that the effect of an update can be propagated through the graph and new updates can be detected.

Separating a learner into distinct classes facilitates designing a graph-based experiment. Each vertex, defined in Algorithm 11, in the graph contains some operation to perform, such as storing a value, calling a method, or setting a property of a factory object. The experiment package is isolated for the rest of the library. Edges connecting vertices act as arguments to a function. For example, Figure 6.1 illustrates calling the create method on the factory graphically. This method throws an exception if the factory cannot create a valid learner object. The graph in the figure is equivalent to the call `Object o = checkFactory(factory, data, iClassAtt);` where the arguments `data` and `iClassAtt` were defined elsewhere in the graph. As shown in Algorithm 11, a Vertex has an output class and multiple inputs. The class of the outputs and inputs is known at configuration time. As a result, we can check that the graph is configured correctly, so if the method call represented by the group were to be written in Java code, it would compile [40].

To implement our goal of adapting to changing configuration, it is necessary for the graph to detect when a portion of it has changed. The graph implements a simple version control system, in which any change in the graph increments a revision number associated with each vertex. As vertices are updated, their revision increases. Thus, at any point in time, each vertex could have a different revision number with the following conditions.

Algorithm 11 Vertex interface, defines an operation to perform in an experiment with multiple inputs and a single output.

```
public interface Vertex {
  public String getKey();
  public long getRevision();
  public Class<?> getOutputClass();
  public void checkArguments(Map<String, Vertex> args)
throws VertexConfigurationException;
}
```

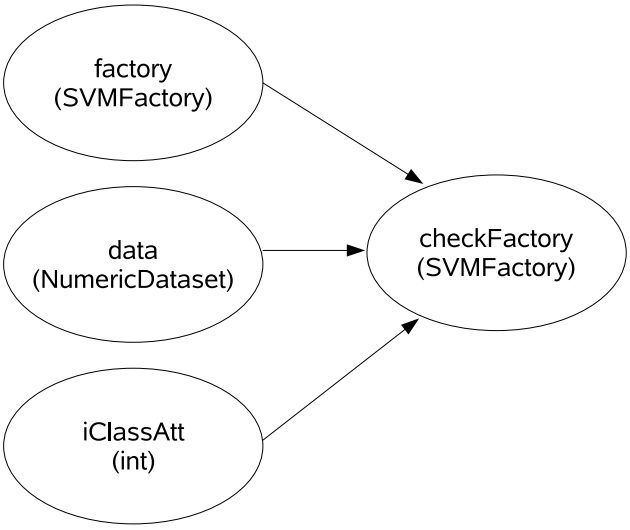


Figure 6.1: Checking a factory's parameters, arguments are denoted by edges.

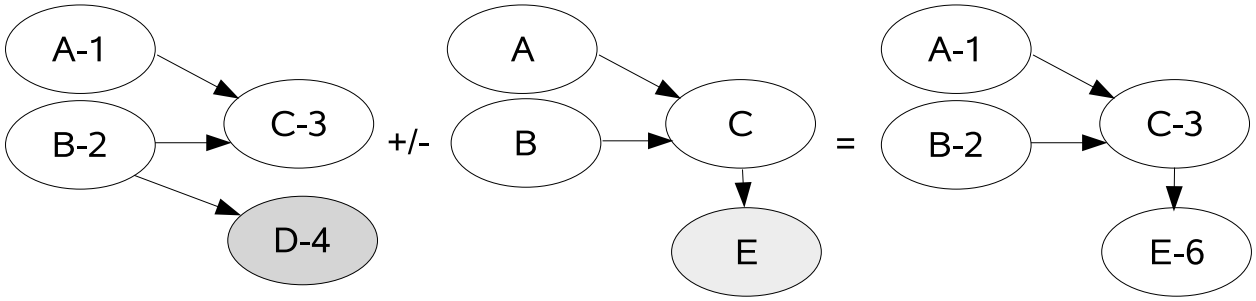


Figure 6.2: Merging versioned graphs. The working copy (unversioned) graph is merged into a versioned graph. Revision numbers are associated with the vertices in the graph and updates are propagated throughout the graph. The final graph's revision number is increased from 4 to 6. Vertices A, B, and C are unchanged because changing outgoing edges does not alter a vertex.

1. If two vertices are connected by a directed edge, the revision of the destination vertex must be greater than or equal to the revision of the source vertex.

2. The revision of the graph is greater than or equal the revision of every vertex.

Building a graph starts with a session very much like standard version control systems like subversion [68]. When an experiment is built, we start with an empty graph with no versioning, called the working copy. Vertices and edges are added to this graph to configure the experiment. After the configuration is complete, the base (versioned) graph is read from a file and the working copy is merged into the base graph. The merging process adds and removes vertices from the base graph that are different from those in the working copy, as illustrated in Figure 6.2. Edges are then changed as appropriate. In the end, the base graph is identical to the working copy, but only those parts that are different in the two graphs have changed. The updated based graph is saved as part of the experiment's configuration.

6.3.2 File Storage

As the experiment is run, only a small portion of the graph will be in use at any given time. For large experiments, keeping references to all previous results is impractical. However, reading results from disk for each part of the experiment is also impractical. Our experiment storage system keeps a weakly-referenced cache of items while storing objects to disk.

Objects generated as part of the experiment are stored as serialized objects, each in a separate file on disk. The name of the file consists of the vertex key and the revision number. From the perspective of a running algorithm, the file storage implements the standard Java Map interface [63]. A weak-reference map stores the objects in memory. When objects are no longer accessible, they can be removed from memory. For related parts of the experiment, most of the access to the storage result in cache hits. In order to avoid multiple access on the same object, each object is copied before it is returned to the caller. Although this increases the storage overhead, it ensures that two running algorithms cannot disrupt each other by altering incoming objects. This maintains the illusion to each object that it is running in a single-threaded context.

Algorithm 12 Example experiment code.

```
1: builder.addGroupKeys("data", "d1", ..., "d5");
2: builder.addGroupValues("data", data);
3: builder.addGroupMethodCall("run_%data_%run",
     DatasetSamplingUtils.class, "shuffle",
     "data_%data", "seed_%run");
4: builder.addGroupMethodCall("model_%data_%run_%alg",
     LearnerExperimentUtils.class, "buildModel",
     "alg_%alg", "run_%data_%run", "iClassAtt");
```

6.3.3 Creating Experiments

Our experiment builder allows the user to create experiments by adding groups of related operations in a single operation. A medium-sized experiment can easily contain $O(10^5)$ vertices and edges. Creating such an experiment manually can quickly become tedious and prone to errors. Experiments are logically organized into groups. We want to load several datasets and then apply several algorithms to them. We then apply some evaluation measures to the results of the algorithms. This can be better expressed as a sequence of operations between groups of vertices in the graph.

The experiment-building process implements a loosely typed language within Java. Vertices are referred to only by their key value. Edges are added between keys and themselves have keys. Vertices contain type information but this is largely transparent to the user, inferred by incident vertices or specified by the user.

In our example experiment, the operations are described as between groups of objects, as shown in Algorithm 12. In line 1 we create an experiment variable, `data`, that identifies a dataset. We then directly create the Dataset objects by associating each element in an array with the key values. as in line 2. This associates each dataset key value with the Dataset object. Now, any time the variable `%data` appears in a vertex key, the key values are substituted. The expression `data_%data` refers to the actual dataset objects. In an experiment a run involves permuting the examples in the dataset, which is implemented in the DatasetSamplingUtils class. We issue the command in line 3, which creates a set of vertices named `run_%data_%run` of size 25, 5 datasets times 5 runs. The key value for each vertex is generated by substituting the keys for each variable. The method `shuffle` in the DatasetSamplingUtils class will be called a total of 25 times. In each method, we take a dataset and a seed value and shuffle the examples in the dataset, returning a new dataset. Now, assuming the algorithms we want to evaluate are defined in the variables `%alg`, we can add the command in line 4 to run all algorithms on all datasets. Here, `iClassAtt` is the constant index of the class attribute for the dataset. With less than two dozen lines of Java code in total, we just created an experiment that has over 1000 steps.

6.3.4 Analyzing Results

After the experiment is complete, or while it is running, we need to be able to analyze the results. The primary analysis tasks are obtaining a revision history and reading results directly from storage, writing analysis code to go through the results and generate figures or reports, and running automated checks to verify the consistency of the experiment.

Revision History

During analysis it is necessary to check whether an object contained in the file is the latest version of that object. A versioned map collects the history of the objects from their file names. Since the name of the file contains the version information, a single directory scan provides the necessary history information. In the analysis, we do not always need to know the revision number for a vertex, requesting only the latest version of the key. Another use for the results is checking that a specific result exists in storage, often to verify specific information about the result.

Writing Analysis Code

In Cougar[2], analysis is typically done offline after the experiment is complete or while it is running. Although some analysis code is better suited to be part of the experiment, that means restarting the experiment whenever the analysis code changes. With the file-based storage system for results, we can often obtain preliminary results while the experiment is running. Thus, most analysis code is written separately from the experiment. We have developed a hybrid analysis method. The analysis code is written as a unit test case, allowing us to run each part separately. The experiment and algorithms are written in Java but the analysis is often done in Java and Matlab [40, 56]. The main advantage of using Matlab for analysis is that figures can be stored in an editable format. It is often the case that prior to publication, tweaks need to be made to the figures, adding titles or arrows, etc. In postscript, this can be extremely hard to achieve by hand. Using Matlab's figure editor, additional titles and comments can be easily added. In addition, statistical analysis tools such as significance testing and advanced plotting features are already implemented. This greatly simplifies the analysis.

Automated Checks

With a large experiment that stores many files, it can be difficult to manually check all aspects of the experiment. Due to the well-defined semantic structure of the experiment graph, automated experiment checkers automate many tedious tasks in experiments.

Before an experiment is run, several checks can be done to determine if the experiment is mis-configured. First of all, it is extremely important to check for cyclic dependencies. These can potentially cause an infinite loop. This is easily accomplished with our graph-based experiment in which we can simply check for cycles in the graph. Although the experiment builder has checks to prevent cycles from entering the graph, it is best to check for them directly. Next, we want to check that the arguments are appropriate. Since each vertex has well-defined operations, we can check that the incoming edges satisfy the requirements. For example, in the small experiment in Figure 6.1, the build vertex is essentially a method call. Using Java reflection, we can determine that the arguments to the method call each need to be an object of the class: Factory, Dataset, and Integer [63]. Since the incident vertices store an output class, all we have to do to check this method call is to check the output class of each incoming vertex. If all these classes are compatible, then the method call will compile if we were to write it in Java.

After the experiment has terminated we do not know if it has terminated because it is finished or because there is an error. By traversing the graph, we can check to see that the results stored on the disk exist, have the right revision, and can be read from the disk without errors. For large experiments, this checking process can take a long time, but is well worth the wait. If the disk runs out of space during the write, some object may not be readable. That part of the experiment would have to be run again, but without this automated check it can be hard to find.

Because the experiment graph and storage keeps all past versions of an object, we need to check that the right version is stored on the disk. We check each object in the storage. There are three possibilities for vertex v with revision s in storage and r in the graph:

1. If vertex v is in the graph and in storage:

   a) If $s < r$, v is obsolete

   b) If $s > r$, an error has occurred in which v is more recent in storage than in the graph.

   c) If $s = r$, v is complete.

2. If v is not in the graph, but is in storage, the v is an orphan:

3. If v is in the graph but not in storage, then v is incomplete.

If v is obsolete or is an orphan, we can safely remove it from storage. The result of this action is to purge the storage of all but the most current vertices. If a vertex is incomplete, then the entire experiment is incomplete.

6.4 Future Work

Although the library has been around for over a year now, much work remains to be done. An important goal is to create better documentation and maintain an open-source environment. New algorithms and bridges to existing packages should be added. A scripting language for experiments would facilitate writing experiments.

A study of open source projects has determined several common features of successful open-source projects [35]. Primarily, in order to bring users from outside the research group into the project, it should be easy for them to participate. Documentation should be easy to access and kept current. Documenting the code is important and there are automated processes for generating this documentation. More important, however, are tutorials describing important features such as writing classifiers, running experiments, and adding new components. A design document with a set of conventions and guidelines for further development is also needed. In addition to merely posting this information, the conventions need to be enforced. A key determinant of a project's survival is the degree to which forums and mailing lists are used to discussed projects. All communications between developers should be posted to the mailing list as much as possible so that new developers have a chance to read the transcripts. Programming issues should be posted and discussed in a well-defined and published manner, making it easy for people not located in the same office to understand what is going on in the project.

Most researchers will continue to develop algorithms in the package or language with which they are most comfortable. An important benefit of our library is that it is easy to design bridges between other packages. Currently, there is support for using WEKA [79] and Matlab [56] algorithms as learners in our library. The WEKA package, being written in Java, is extremely easy to support. For Matlab, a C-library has been written to call Matlab as efficiently as possible. It is, however, much easier to call Java from Matlab rather than Matlab from Java. Future work should make it easier to call Java from Matlab with specialized functions for converting objects and an easy way to add all necessary library dependencies.

Finally, creating and running experiments has become the most challenging aspect of using the software. This is due, in part, to the complexity of the experiments that we run. It is also due to the complexity of versioned graphs and storing intermediate results. A scripting language, based on an existing functional language would greatly facilitate this process. The latest edition of Java, version 6 [40], adds support for scripting languages. In order to use scripting, the language would have to support grouped operations and a graph structure.

6.4.1 Vision for Cougar[2]

The software library is just one component for a better approach to developing algorithms as part of research activities. It is our intention to make data mining and machine learning research an open-source activity. This will require a single repository for algorithms and datasets. Researchers should be able to add whatever algorithms they use to the library. Experiments built in the library can be stored for later use and comparison with other work. Instead of implementing an algorithm from a paper, we can simply call the algorithm in the library. If there are problems with the code, the authors are available for issue tracking and respond to the mailing list. By bringing transparency to the research process, errors in algorithms can be quickly discovered and resolved. This increases the pace of research allowing new ideas to be quickly implemented and evaluated.

7 Conclusion

This dissertation presented a new theoretical and algorithmic learning framework called multi-objective multi-task learning. The key contributions of this work to the existing body of literature are:

- Formulation of the empirically efficient learning and the multi-objective multi-task learning frameworks (Chapter 1).
- New sample complexity bounds for weighted objectives and multiple loss functions (Chapter 2).
- Application of the learning principle to support vector machines and distance function learning (Chapter 3).
- Cougar^2: an open-source library for machine learning and data mining (Chapter 6).

The work is significant in the existing body of literature in three main respects. First, new theoretical results are derived, showing that multi-objective multi-task learning generalizes both single-task and single-objective multi-task learning. Second, new algorithms are proposed that illustrate how the theory can be applied in practical algorithms. Finally, a new software library is developed that contains not only the present work but allows developers in the community to develop new types of algorithms.

7.1 Learning Frameworks

In Chapter 2, we discussed three different learning frameworks for multi-objective multi-task learning. First, we considered a weighted combination of objectives where the weight vector is known. Second, we considered finding the empirically efficient set given multiple data-dependent loss functions. Finally, we considered tradeoffs among clusters of similar tasks.

Most existing single-objective multi-task learning algorithms already balance multiple objectives. Methods such as cross-validation or regularization fail to utilize critical concepts from multi-objective optimization, such as Pareto optimality or efficient solutions. The true loss function in these algorithms is actually a vector-valued function coupled with a weight vector. We showed that the weights play an important role in bounding the sample complexity of the learner.

Our second framework directly considers the problem of empirically efficient learning, finding the empirical tradeoff surface of multiple data-dependent loss functions. Convergence is defined in terms of the Hausdorff distance to the expected tradeoff surface. The problem was reduced to bounding a vector-valued loss function. We then showed that the sample complexity increases, in the worst case, linearly with the number of objectives.

The last framework considers clusters of related tasks in which there may be no hypothesis space that minimizes loss for all clusters. In that case, we seek a Pareto optimal hypothesis space with respect to the average loss of each cluster of tasks.

All of the sample complexity results depend on the formulations of Baxter and Maurer [13, 58]. Most of the complexity results are data dependent and thus depend on both the learner and the dataset.

7.2 Algorithms

The key to designing multi-objective multi-task algorithms is deciding what objectives to optimize. Each algorithm in Chapter 3 optimizes different objectives, designed for one of our learning frameworks. Three algorithms extend an existing multi-task SVM algorithm. The fourth algorithm learns a common set of weights for the feature vector.

The first support vector algorithm (SVM) is a straightforward extension to the existing work in which regularization is posed as a multi-objective problem. The regularization terms become additional objectives. The tradeoff surface helps us select an efficient solution that minimizes expected loss.

The second support vector algorithm, SVMTPFP, minimizes the average loss in each class as a separate objective. Minimizing the false-negative and false-positive rates, equivalently maximizing the true-positive and minimizing the false positive rates, results in tradeoff surfaces that resemble ROC curves. The key difference is that each point corresponds to a unique hypothesis space rather than a threshold value.

The third algorithm, SVMGroup, assumes an inherent clustering in the tasks. One common model is shared by the tasks, but it may not be possible to fit all clusters equally well. The tradeoffs tell us how different clusters respond to the set of efficient hypothesis spaces.

The distance function learning algorithm (DFL) is a convex formulation of a common iterative algorithm. It learns a weight for each feature. A 1-nearest-neighbor classifier uses the resulting weighted distance function to classify incoming examples. The two objectives are to minimize the distance between an example and its nearest neighbor in the same class and maximize the separation between examples in different classes. The tradeoff surface shows that these objectives are not always compatible.

7.3 Applications

Multi-objective multi-task learning allows many more applications to be solved than is possible with single-objective multi-task learning algorithms. Although the present work demonstrates the potential of the approach, we have only begun to apply it to real problems. This section discusses potential application areas and how the proposed learning frameworks would be utilized.

7.3.1 Recommender Systems

There are two important application of multi-objective multi-task learning for recommender systems. Both assume that there is a vector of features that describes a product in a catalog. A consumer is associated with a history, a dataset consisting of products and actions taken (e.g., purchase, add to cart, etc.). We want to predict whether the consumer purchases a product. Each consumer's history is modeled as a learning task, so that we ideally want to learn a separate hypothesis for each consumer.

Our first goal is to send recommendations to consumers for products that he or she might want to purchase. Our transferred knowledge from other consumers allows us to make reasonable recommendations to a new consumer even without having many examples. The multiple loss functions framework can be applied here where we would balance the true-positive and false-positive rates. We want to send recommendations to consumers that we expect to purchase the product and not to consumers that we do not expect to purchase the product. The tradeoff surface will allow us to select a balance between the two objectives that can apply to all consumers. This is in contrast to the ROC curve where we would have to select a separate threshold for each consumer.

In our proposed task clustering framework, consumers could belong to clusters of consumers, known in the literature as market segments. For example, we can cluster consumers by products for which they have similar preferences. This may lead to clusters called hikers, who prefer hiking equipment, and audiophiles, who prefer to buy music from a specific genre. We may not have many examples for each consumer, but

multi-task learning allows us to transfer knowledge across consumers. So, by learning a common hypothesis space for both hikers and audiophiles, we reuse our knowledge across all consumers but can prefer lower loss in one cluster versus another. For example, suppose that we obtain a higher profit margin selling hiking equipment than music. Thus, we cannot tolerate a high error rate for hiking equipment, because it could lead to lower revenue. In analyzing the tradeoff surface, we would see that this scheme obtains lower error for hikers at the expense of the audiophiles, simply because hikers are more important to our bottom line. We could separate these clusters and learn separate hypothesis spaces for each cluster, but no knowledge would be transferred between the clusters under this scheme. This could result in duplicating work in both clusters.

7.3.2 Decision Automation

Standard decision theory allows us to quantify the effect of a simple decision, leading us to choose the action with the lowest expected loss. Often, the effect of the decision has outcomes with multiple, conflicting loss. Multi-objective methods would lead us to choose a Pareto-optimal (efficient) action. In decision automation, a learning algorithm arrives at a decision through classification. The outcome of the decision can have many different costs and benefits.

Suppose we want to predict whether a consumer's credit card has been stolen based on a profile of past behavior. If we claim the card has been stolen and it was not, the consumer could become angry. Numerous effects could occur, such as the consumer canceling the service or stop using the card. If we model the consumer's past behavior as a learning task, then there is a cost associated with the decision that is dependent on the learning task. We could then cluster the consumers based on similar reactions. This leads us to what are commonly called risk pools in insurance [78]. Our proposed task clustering framework can be applied here to analyze the tradeoff between groups. We can find a hypothesis space that, for example, has lower error for the "quick-to-leave" consumer–who does not tolerate errors–at the expense of increasing error for the "better-safe-than-sorry" consumer–who would tolerate some errors.

Alternatively, we could make the decision to deny a single transaction based on whether it is consistent with a consumer's profile. If the transaction is denied, we lose the potential interest accrued by the transaction. For example, a large purchase may accrue a large daily finance charge, resulting in profit for the credit card company. Thus, the features of the transaction itself will influence our decision, so our loss function depends on each example. The tradeoff surface will allow us to see, in monetary terms, the effect of the different efficient hypothesis spaces. Thus, we can customize the learning algorithm to take into account these different loss functions, more so than is possible with existing approaches.

7.4 Limitations

There are a few limitations to the proposed work. Most of the limitations relate to the algorithms or the implementation and not specifically to the proposed empirically efficient learning principle.

The loss functions, as they appear in the algorithms, are defined over the entire dataset. Many applications require more fine-grained loss functions. Although this case is covered in the theoretical framework, new algorithms will have to be designed.

The learning algorithms we propose are defined only for two objectives. This is not a limitation of the learning principle or the learning algorithm. The algorithms were limited to two objectives because we were not able to obtain an implementation of multi-objective optimization algorithm for more than two objectives. Few practical algorithms are available and most are not easily adapted to our learning problem. The current approach is very similar to the cross-validation approach, but this is not an inherent limitation in the framework.

The SVM algorithms have not been kernelized. Although the extension has been pursued in multi-task learning, it is still very much an open problem. The original feature space is sufficient for our purposes, to demonstrate the new multi-objective learning frameworks.

The current implementation of the algorithms does not scale to large datasets. This is due to the Matlab implementation and the employed optimization library. The current approach was adopted because it allowed rapid development of the algorithms, which offered proof-of-concept results. A scalable algorithm can be designed, but it is left for future work.

The theoretical results make rather strong assumptions on the boundedness of the loss functions and the distance between tradeoff surfaces, which leads to loose upper bounds. Our goal was not to obtain extremely tight bounds but to demonstrate the asymptotic relationship between the different learning frameworks.

7.5 Future Work

The present work lays the foundations for a new research area into multi-objective multi-task learning. Much work remains to be done, however. First, we must overcome key practical challenges such as efficient optimization algorithms and evaluation methods. We can then explore new application areas and extensions to the ideas.

The algorithms, relying on convex optimization, can be slow to run for large datasets. In future work, we can consider gradient-based approaches or evolutionary algorithms to improve speed. Gradient-based approaches have been used for related algorithms with some success. The key for multi-objective optimization is to rely on the connectedness of the efficient set, guaranteed by the convexity of the problem. Thus, we can start from any efficient point and enumerate the tradeoff surface directly.

Although the tradeoff surface has been shown to be useful, it is difficult to visualize higher dimensional surfaces. We could instead focus on learning objective weights from extra information about the dataset or from users directly. We could also employ dimension-reduction methods or manifold learning to reduce the multi-dimensional surface to 2 or 3 dimensions.

Each of the proposed algorithms optimizes a different, incomparable, set of objectives. Future work should consider a standard set of objectives in order to permit a more appropriate comparison than average accuracy. Although expected loss is important, it reduces multi-objective learning to a single-objective case, which misses the point of the framework.

The theoretical results consist mostly of one-sided bounds and rely on a reduction in each case to the single-objective framework. This potentially inflates the upper bound. It would be interesting to investigate convergence within the Hausdorff metric space. This would require a covering of the set of tradeoff surfaces, expectations over random sets, and one would expect it to be fairly complex.

Bibliography

[1] Shivani Agarwal, Thore Graepel, Ralf Herbrich, Sariel Har-Peled, and Dan Roth. Generalization bounds for the area under the roc curve. *J. Machine Learning Research*, 6:393–425, 2005.

[2] Rie Kubota Ando and Tong Zhang. A framework for learning predictive structures from multiple tasks and unlabeled data. *J. Machine Learning Research*, 6:1817–1853, 2005.

[3] Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge UP, New York, NY, USA, 1999.

[4] Andreas Argyriou, Theodoros Evgeniou, and Massimiliano Pontil. Multi-task feature learning. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Adv. in Neural Information Processing Systems 19*, pages 251–258, Cambridge, MA, 2007. MIT Press.

[5] Abraham Bagherjeiran. Artificial government: Meta-learning for multi-agent coordination and control. Master's thesis, University of Houston, 2005.

[6] Abraham Bagherjeiran, Christoph F. Eick, Chun-Sheng Chen, and Ricardo Vilalta. Adaptive clustering: Obtaining better clusters using feedback and past experience. In *Proc. 5th Int'l Conf. on Data Mining*, pages 565–568, Houston, TX, USA, November 2005. IEEE Computer Society.

[7] Abraham Bagherjeiran, Ricardo Vilalta, and Christoph F. Eick. Content-based image retrieval through a multi-agent meta-learning framework. In *Proc. 17th Int'l Conf. on Tools with Artificial Intelligence*, pages 24–28, Hong Kong, China, November 2005. IEEE Computer Society.

[8] Bram Bakker and Tom Heskes. Task clustering and gating for bayesian multitask learning. *J. Machine Learning Research*, 4:83–99, 2003.

[9] Arindam Banerjee. On bayesian bounds. In *Proc. 23rd Int'l Conf. on Machine Learning*, pages 81–88, 2006.

[10] Peter L. Bartlett, Stéphane Boucheron, and Gábor Lugosi. Model selection and error estimation. *Machine Learning*, 48(1-3):85–113, 2002.

[11] Peter L. Bartlett, Olivier Bousquet, and Shahar Mendelson. Local rademacher complexities. *Annals of Statistics*, 33:1497–1537, 2005.

[12] Peter L. Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *J. Machine Learning Research*, 3:463–482, 2002.

[13] Jonathan Baxter. A model of inductive bias learning. *J. Artificial Intelligence Research*, 12:149–198, 2000.

[14] Shai Ben-David and Reba Schuller. Exploiting task relatedness for multiple task learning. In *Proc. 16th Conf. on Learning Theory*, pages 567–580, Washington, DC, USA, August 2003.

[15] Hilan Bensusan and Christophe G. Giraud-Carrier. Discovering task neighbourhoods through landmark learning performances. In *Proc. 4th European Conf. on Principles of Data Mining and Knowledge Discovery*, pages 325–330, London, UK, 2000. Springer-Verlag.

[16] Helmut Berrer, Iain Paterson, and Jörg Keller. Evaluation of machine-learning algorithm ranking advisors. In *Workshop on Data Mining, Decision Support, Meta-learning and ILP: Forum for Practical Problem. In Proc. 5th European Conf. on Principles of Data Mining and Knowledge Discovery*, 2001.

[17] Jinbo Bi. Multi-objective programming in svms. In *Proc. 20th Int'l Conf. on Machine Learning*, pages 35–42, 2003.

[18] Olivier Bousquet, Stéphane Boucheron, and Gábor Lugosi. Introduction to statistical learning theory. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning*, number 3176 in LNAI, pages 169–207. Springer-Verlag, 2003.

[19] Olivier Bousquet and André Elisseeff. Stability and generalization. *J. Machine Learning Research*, 2:499–526, 2002.

[20] Pavel B. Brazdil, Carlos Soares, and Joaquim Pinto da Costa. Ranking learning algorithms: Using IB1 and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[21] Dmitri Burago, Yuri Burago, and Sergei Ivanov. *A Course in Metric Geometry*, volume 33 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, USA, 2001.

[22] Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[23] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. In *IEEE Trans. on Information Theory*, volume 13, pages 21–27, 1967.

[24] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, 1991.

[25] Lorcan Coyle and Pádrain Cunningham. Improving recommendation ranking by learning personal feature weights. In *Proc. 7th European Conf. on Adv. in Case-Based Reasoning*, LNAI 3155, pages 560–572, Madrid, Spain, September 2004.

[26] Deb. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. 2000.

[27] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, New York, NY, USA, 2001.

[28] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, New York, 2nd edition, 2001.

[29] Matthias Ehrgott. *Multicriteria Optimization*. Springer, New York, NY, USA, 2nd edition, 2005.

[30] Christoph F. Eick, Alain Rouhana, Abraham Bagherjeiran, and Ricardo Vilalta. Using clustering to learn distance functions for supervised similarity assessment. In P. Perner and A. Imiya, editors, *Proc. 4th Int'l Conf. Machine Learning and Data Mining in Pattern Recognition*, LNAI 3587, pages 120–131, Berlin, 2005. Springer Verlag.

[31] Richard M. Everson and Jonathan E. Fieldsend. Multi-class roc analysis from a multi-objective optimisation perspective. *Pattern Recognition Letters*, 27(8):918–927, 2006.

[32] Theodoros Evgeniou, Charles A. Micchelli, and Massimiliano Pontil. Learning multiple tasks with kernel methods. *J. Machine Learning Research*, 6:615–637, 2005.

[33] Theodoros Evgeniou and Massimiliano Pontil. Regularized multi-task learning. In *Proc. 10th Int'l Conf. on Knowledge Discovery and Data Mining*, pages 109–117, Seattle, Washington, USA, August 2004.

[34] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[35] Karl Fogel. *Producing Open Source Software: How to run a successful free software project*. O'Reilly, Sebastopol, CA, USA, 2006.

[36] Apache Software Foundation. *Apache Ant (Vers. 1.7.0)*, December 2006. Accessed April 2007 http://ant.apache.org/.

[37] Holger Fröhlich and Andreas Zell. Efficient parameter selection for support vector machines in classification and regression via model-based global optimization. In *Proc. Int'l Joint Conf. on Neural Networks*, pages 1431–1438, 2005.

[38] Johannes Fürnkranz and Johann Petrak. An evaluation of landmarking variants. In C. Giraud-Carrier, N. Lavra, Steve Moyle, and B. Kavsek, editors, *Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning in 12th European Conf. on Machine Learning*, 2001.

[39] Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. An information theoretic tradeoff between complexity and accuracy. In *Proc. 16th Conf. on Learning Theory*, pages 595–609, 2003.

[40] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, Upper Saddle River, NJ, USA, 3 edition, 2005.

[41] Michael Grant, Stephen Boyd, and Yinyu Ye. *Nonconvex Optimization and Its Applications*, volume 84, chapter Disciplined Convex Programming, pages 155–210. Springer, 2006.

[42] David J. Hand and Robert J. Till. A simple generalisation of the area under the roc curve for multiple class classification problems. *Machine Learning*, 45(2):171–186, 2001.

[43] Trevor Hastie and Robert Tibshirani. Discriminant adaptive nearest neighbor classification and regression. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Adv. in Neural Information Processing Systems 8*, pages 409–415. The MIT Press, 1996.

[44] Tony Jebara. Multi-task feature and kernel selection for svms. In *Proc. 21st Int'l Conf. on Machine Learning*, pages 55–62, 2004.

[45] Thorsten Joachims. A support vector method for multivariate performance measures. In *Proc. 22nd Int'l Conf. on Machine Learning*, pages 377–384, 2005.

[46] Brendan Juba. Estimating relatedness via data compression. In *Proc. 23rd Int'l Conf. on Machine Learning*, pages 441–448, 2006.

[47] Michael Kearns, Yishay Mansour, Andrew Y. Ng, and Dana Ron. An experimental and theoretical comparison of model selection methods. *Machine Learning*, 27:7–50, 1997.

[48] Seung-Jean Kim, Alessandro Magnani, Sikandar Samar, Stephen Boyd, and Johan Lim. Pareto optimal linear classification. In *Proc. 23rd Int'l Conf. on Machine Learning*, pages 473–480, 2006.

[49] Vladimir Koltchinskii. Rademacher penalties and structural risk minimization. *IEEE Trans. Information Theory*, 47(5):1902–1914, 2001.

[50] Igor Kononenko. Estimating attributes: Analysis and extensions of RELIEF. In *Proc. 7th European Conf. on Machine Learning*, pages 171–182, 1994.

[51] Rui Leite and Pavel Brazdil. Improving progressive sampling via meta-learning on learning curves. In *Proc. 15th European Conf. on Machine Learning*, pages 250–261, 2004.

[52] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitányi. The similarity metric. In *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, pages 863–872, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[53] Xuejun Liao and Lawrence Carin. Radial basis function network for multi-task learning. In *Adv. in Neural Information Processing Systems 18*, pages 628–635, 2005.

[54] Lawrence Mandow and José-Luis Pérez de-la Cruz. A new approach to multiobjective a* search. In *IJCAI*, pages 218–223, 2005.

[55] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[56] Mathworks. Matlab (vers. 7 r14), 2004. Accessed April 2007 http://www.mathworks.com/.

[57] Andreas Maurer. Algorithmic stability and meta-learning. *J. Machine Learning Research*, 6:967–994, 2005.

[58] Andreas Maurer. Bounds for linear multi-task learning. *J. Machine Learning Research*, 7:117–139, 2006.

[59] David A. McAllester. PAC-bayesian stochastic model selection. *Machine Learning*, 51(1):5–21, 2001.

[60] James M. McQueen. Some methods of classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297, 1967.

[61] Charles A. Micchelli and Massimiliano Pontil. A function representation for learning in banach spaces. In *Proc. 18th Conf. on Learning Theory*, pages 255–269, 2004.

[62] Charles A. Micchelli and Massimiliano Pontil. Kernels for multi–task learning. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Adv. in Neural Information Processing Systems 17*, pages 921–928. MIT Press, Cambridge, MA, 2004.

[63] Sun Microsystems. *Java (Vers. 1.5)*, 2005. Accessed April 2007 http://java.sun.com/.

[64] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, and Scholz. Yale: Rapid prototyping for complex data mining tasks. In Ralf Klinkenberg, Stefan Rüping, Andreas Fick, Nicola Henze, Christian Herzog, Ralf Molitor, and Olaf Schröder, editors, *Proc. 12th Int'l Conf. on Knowledge Discovery and Data Mining*, number 763 in Forschungsberichte des Fachbereichs Informatik, Universität Dortmund, pages 84–92, Dortmund, Germany, 10 2006. ISSN 0933-6192.

[65] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[66] Iain Paterson and Helmut Berrer Jörg Keller. The focused multi-criteria ranking approach to machine learning algorithm selection - an incremental meta learning assistant for data mining tasks. In *Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-learning in 12th European Conf. on Machine Learning*, pages 93–108, 2001.

[67] Michael J. Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial Intelligence Review*, 13(5-6):393–408, 1999.

[68] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, Sebastopol, CA, USA, 2004. April 2007 http://svnbook.org/.

[69] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, USA, 2002.

[70] Dale Schuurmans and Finnegan Southey. Metric-based methods for adaptive model selection and regularization. *Machine Learning*, 48(1-3):51–84, 2002.

[71] John Shawe-Taylor and Peter L. Bartlett. Structural risk minimization over data-dependent hierarchies. *IEEE Trans. Information Theory*, 44(5):1926–1940, 1998.

[72] Carlos Soares, Pavel Brazdil, and Petr Kuba. A meta-learning method to select the kernel width in support vector regression. *Machine Learning*, 54(3):195–209, 2004.

[73] Jos F. Sturm. Using SeDuMi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11:625–653, 1999. Software available at: http://sedumi.mcmaster.ca/.

[74] Ambuj Tewari and Peter L. Bartlett. On the consistency of multiclass classification methods. In *Proc. 18th Conf. on Learning Theory*, pages 143–157, 2005.

[75] Sebastian Thrun and Joseph O'Sullivan. Clustering learning tasks and the selective cross-task transfer of knowledge. In *Learning to Learn*. Kluwer Academic Publishers, 1998.

[76] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York, NY, USA, 2nd edition, 2000.

[77] Kilian Weinberger, John Blitzer, and Lawrence Saul. Distance metric learning for large margin nearest neighbor classification. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Adv. in Neural Information Processing Systems 18*, pages 1473–1480. MIT Press, Cambridge, MA, 2005.

[78] Charles Wilson. A model of insurance markets with incomplete information. *J. Economic Theory*, 16:167–207, 1977.

[79] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools with Java implementation*. Morgan Kaufmann, San Francisco, 2 edition, 2005.

[80] Ya Xue, Xuejun Liao, Lawrence Carin, and Balaji Krishnapuram. Multi-task learning for classification with dirichlet process priors. *J. Machine Learning Research*, 8:35–63, 2007.

[81] Kai Yu and Volker Tresp. Learning to learn and collaborative filtering. In *Workshop on Inductive Transfer: 10 Years Later in Adv. in Neural Information Processing Systems 18*, 2005.

[82] Kai Yu, Volker Tresp, and Anton Schwaighofer. Learning gaussian processes from multiple tasks. In *Proc. 22nd Int'l Conf. on Machine learning*, pages 1012–1019, New York, NY, USA, 2005. ACM Press.

[83] Jian Zhang, Zoubin Ghahramani, and Yiming Yang. Learning multiple related tasks using latent independent component analysis. In *Adv. in Neural Information Processing Systems 17*, pages 435–442. 2005.

[84] Eckart Zitzler, Marco Laumanns, and Stefan Bleuler. A tutorial on evolutionary multiobjective optimization. In *Workshop on Multiple Objective Metaheuristics*, pages 3–38. Springer-Verlag, Berlin, Germany, 2004.

[85] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. Technical report, Institut für Technische Informatik und Kommunikationsnetze, ETH Zurich, 2002.