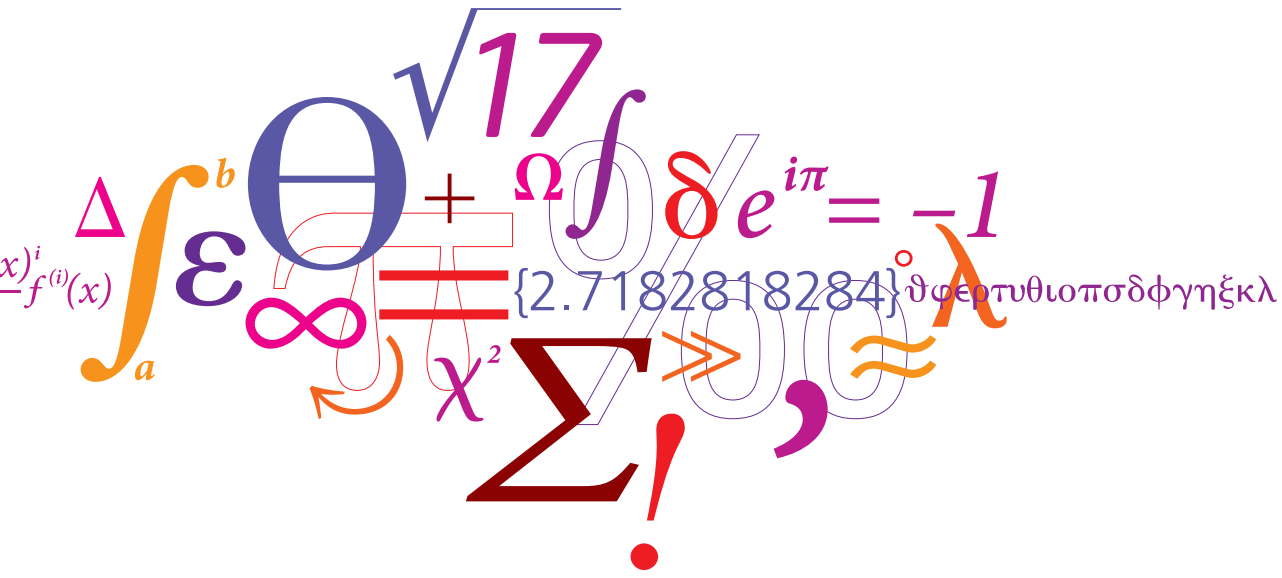
DTU Technical University of Denmark

Master's Thesis

Concepts in Predictive Machine Learning

A conceptual framework for approaching predictive modelling problems and case studies of competitions on Kaggle.



Author
David Kofoed Wind s082951

Supervisor
Ole Winther
PhD, Associate Professor

Abstract

Predictive modelling is the process of creating a statistical model from data with the purpose of predicting future behavior. In recent years, the amount of available data has increased exponentially and "Big Data Analysis" is expected to be at the core of most future innovations. Due to the rapid development in the field of data analysis, there is still a lack of consensus on how one should approach predictive modelling problems in general.

Another innovation in the field of predictive modelling is the use of data analysis competitions for model selection. This competitive approach is interesting and seems fruitful, but one could ask if the framework provided by for example Kaggle gives a trustworthy resemblance of real-world predictive modelling problems.

In this thesis, we will state and test a set of hypotheses about predicitive modelling, both in general and in the scope of data analysis competitions. We will then describe a conceptual framework for approaching predictive modelling problems. To test the validity and usefulness of this framework, we will participate in a series of predictive modelling competitions on the platform provided by Kaggle, and describe our approach to these competitions.

Due to the diversity in the competitions at Kaggle combined with the breadth of data analysis approaches, we can not state a perfect algorithm for competing on Kaggle. Still we think that by utilizing the lessons learnt by previous competitors and combining this with some of the tricks described in this thesis, one can without too much previous experience obtain decent results.

Preface

This thesis was prepared in the Section for Cognitive Systems, DTU Compute at the Technical University of Denmark in the period from September 2013 to March 2014. It corresponds to 30 ECTS credits, and it was part of the requirements for acquiring an M.Sc. degree in engineering.

I would like to thank my supervisor Ole Winther for careful guidance, patience and support, without him this thesis would not be what it is. I would also like to thank my parents for supporting my education, and my friends for listening to my mathematical rambling and supporting me when the code was crashing.

I would like to thank Pia Wind for proofreading and Rasmus Malthe Jørgensen for proofreading and for help with some C++. Additionally, I would like to thank Dan Svenstrup for countless interesting discussions about Kaggle, Machine Learning and additional interesting topics.

David Kofoed Wind

Contents

Chapter 1

Introduction

In recent years, the amount of available data has increased exponentially and "Big Data Analysis" is expected to be at the core of most future innovations [Lohr, 2012, Manyika et al., 2011, Forum, 2012]. A new and very promising trend in the field of predictive machine learning is the use of data analysis competitions for model selection. Due to the rapid development in the field of competitive data analysis, there is still a lack of consensus and literature on how one should approach predictive modelling competitions.

1.1 Algorithmic modelling

In his well-known paper "Statistical Modeling : The Two Cultures" [Breiman, 2001b], Leo Breiman divides statistical modelling into two cultures, the *data modelling culture* and the *algorithmic modelling culture*, visualized in Figure 1.1:

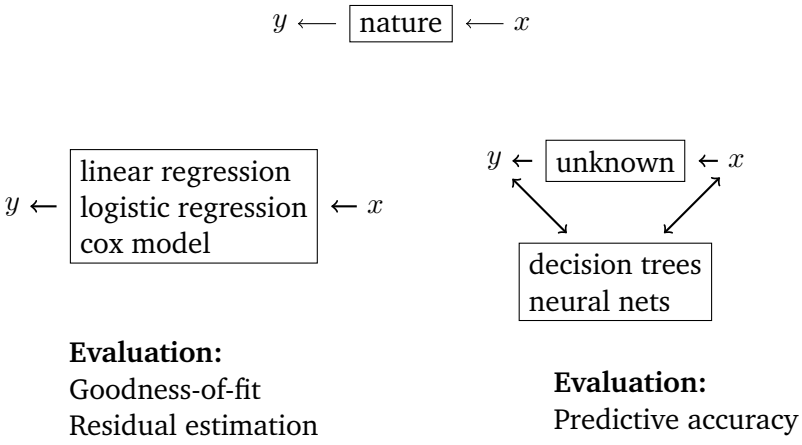


Figure 1.1: The top part of the figure is the way Breiman pictures the world as a model taking input x and outputting y. In the bottom part, the two approaches to data analysis are sketched, on the left data modelling and on the right algorithmic modelling. In algorithmic modelling, we do not try to infer nature's true model.

Breiman argues that classical statistics (the data modelling culture) relies too much on modelling data, and too little on making accurate models for prediction. He argues that predictive accuracy should be the main determiner of how well a statistical model performs:

> "The most obvious way to see how well the model box emulates nature's box is this: put a case x down nature's box getting an output y. Similarly, put the same case x down the model box getting an output y'. The closeness of y and y' is a measure of how good the emulation is. For a data model, this translates as: fit the parameters in your model by using the data, then, using the model, predict the data and see how good the prediction is. Prediction is rarely perfect. There are usually many unmeasured variables whose effect is referred to as "noise". But the extent to which the model box emulates nature's box is a measure of how well our model can reproduce the natural phenomenon producing the data.

Another point put forward in [Breiman, 2001b] is that complex and non-interpretable models almost always outperforms simple interpretable models when it comes to predictive accuracy. The argument for why this is to be expected is that noise in the data leads to unstable/non-robust results. In the book [McCullagh and Nelder, 1989] they write that

> "Data will often point with almost equal emphasis on several possible models, and it is important that the statistician recognize and accept this.

Breiman describes this effect as the Rashomon Effect:

> Rashomon is a wonderful Japanese movie in which four people, from different vantage points, witness an incident in which one person dies and another is supposedly raped. When they come to testify in court, they all report the same facts, but their stories of what happened are very different. What I call the Rashomon Effect is that there is often a multitude of different descriptions (equations $f(x)$) in a class of functions giving about the same minimum error rate.

He then argues that aggregation of simple models (also known as ensembling) leads to better predictions at the cost of lowering interpretability. He proceeds to give numerical evidence for this tendency by comparing a decision tree to a random forest on a number of publicly available datasets, these results can be seen in Table 1.1.

Dataset	**Forest**	**Single tree**
Breast cancer	2.9	5.9
Ionosphere	5.5	11.2
Diabetes	24.2	25.3
Glass	22.0	30.4
Soybean	5.7	8.6
Letters	3.4	12.4
Satellite	8.6	14.8
Shuttle	7.0	62.0
DNA	3.9	6.2
Digit	6.2	17.1

Table 1.1: The test set misclassification error (%) on 10 different datasets from the UCI repository (http://archive.ics.uci.edu/ml/).

From the results shown in Table 1.1, it is clear that the complex but un-interpretable models achieves a lower prediction error – consequently underlining the reason for transcending to algorithmic modelling as the core philosophy.

The arguments put forward in [Breiman, 2001b] justifies an approach to predictive modelling where the focus is purely on predictive accuracy. That this is the right way of looking at statistical modelling is the underlying assumption in statistical prediction competitions, and consequently also in this thesis.

1.2 Data prediction competitions

A recent innovation in the field of predictive modelling is the use of predictive modelling competitions for model selection. This concept was made popular with the Netflix Prize, a massive open competition with the aim of constructing the best algorithm

for predicting user ratings of movies. The competition featured a prize of 1,000,000 dollars for the first team to improve Netflix's own results by 10%. After the success with the Netflix Prize, the platform Kaggle was born, providing a platform for predictive modelling. Kaggle hosts numerous data prediction competitions and has more than 150,000 users worldwide.

The basic structure of a predictive modelling competition – as seen for example on Kaggle and in the Netflix competition – is the following: A predictive problem is described, and the participants are given a dataset with a number of samples and the true target values (the values to predict) for each sample given, this is called the training set. The participants are also given another dataset like the training set, but where the target values are not known, this is called the test set. The task of the participants is to predict the correct target values for the test set, using the training set to build their models. When participants have a set of proposed predictions for the test set, they can submit these to a website, which will then evaluate the submission on a part of the test set known as the quiz set, the validation set or simply as the public part of the test set. The result of this evaluation on the quiz set is shown in a leaderboard giving the participants an idea of how they are progressing.

Using a competitive approach to predictive modelling is being praised by some as the modern way to do science:

> "Kaggle recently hosted a bioinformatics contest, which required participants to pick markers in a series of HIV genetic sequences that correlate with a change in viral load (a measure of the severity of infection). Within a week and a half, the best submission had already outdone the best methods in the scientific literature. [Goldbloom, 2010]
>
> (Anthony Goldbloom, Founder and CEO at Kaggle)

> "These prediction contests are changing the landscape for researchers in my area — an area that focuses on making good predictions from finite (albeit sometimes large) amount of data. In my personal opinion, they are creating a new paradigm with distinctive advantages over how research is traditionally conducted in our field. [Mu, 2011]
>
> (Mu Zhu, Associate Professor, University of Waterloo)

This competitive approach is interesting and seems fruitful – one can even see it as an extension of the aggregation ideas put forward in [Breiman, 2001b]. Still one should ask if the framework provided by for example Kaggle gives a trustworthy resemblance of real-world predictive modelling problems where problems do not come with a quiz set and a leaderboard.

Thesis outline In Chapter 2 we will state and investigate a set of hypotheses about predictive modelling in a competitive framework. In Chapter 3 we will outline a con-

ceptual framework for approaching predictive modelling competitions – from the initial data preprocessing to final model selection and combination. This framework will be built on personal experience, literature studies, interviews we did with prominent data analysts and on mathematical theory. Chapter 4 describes the underlying mathematical theory behind the models used throughout the thesis. In Chapter 5 and Chapter 6 we describe two specific competitions on the Kaggle platform, our approach in the competitions and the results obtained. This will help validate our conceptual framework and will put the hypotheses to another test. Finally we conclude in Chapter 7.

Chapter 2

Hypotheses

In this chapter we state 5 hypotheses about predictive modelling in a competitive framework. We will try to verify the validity of each hypothesis using a combination of mathematical arguments, empirical evidence from previous competitions (gathered in Appendix A) and qualitative interviews we did with some of the top participants at Kaggle (gathered in Appendix C).

The five hypotheses to be investigated in this chapter are:

1. Feature engineering is the most important part of predictive machine learning
2. Overfitting to the leaderboard is a real issue
3. Simple models can get you very far
4. Ensembling is a winning strategy
5. Predicting the right thing is important

2.1 Feature engineering is the most important part

With the extensive amount of free tools and libraries available for data analysis, everybody has the possibility of trying advanced statistical models in a competition. As a consequence of this, what gives you most "bang for the buck" is rarely the statistical method you apply, but rather the features you apply it to.

> "For most Kaggle competitions the most important part is feature engineering, which is pretty easy to learn how to do. (Tim Salimans)

> “As a general rule good feature selection is what determines success rather than clever algorithms.
>
> Response to question on the Kaggle Forum:
> http://www.kaggle.com/forums/t/5762/what-is-your-workflow/31264 (Robin East)

There are some types of data where feature engineering matters more. Examples of such data types is natural language data and image data. In many of the previous competitions with text data and image data, feature engineering was a huge part of the winning solutions (examples of this are for example SUNNYHASHTAGS, FACEBOOK, SEECLICKPREDICT and BIRD).

In the competition SUNNYHASHTAGS (described in Appendix A.2) which featured text data taken from Twitter, feature engineering was a major part of the winning solution. The winning solution used a simple regularized regression model (described in Section 4.3), but generated a lot of features from the text:

> “My set of features included the basic tfidf of 1,2,3-grams and 3,5,6,7 ngrams. I used a CMU Ark Twitter dedicated tokenizer which is especially robust for processing tweets + it tags the words with part-of-speech tags which can be useful to derive additional features. Additionally, my base feature set included features derived from sentiment dictionaries that map each word to a positive/neutral/negative sentiment. I found this helped to predict S categories by quite a bit. Finally, with Ridge model I found that doing any feature selection was only hurting the performance, so I ended up keeping all of the features ˜ 1.9 mil. The training time for a single model was still reasonable. (aseveryn - 1st place winner)

In the competitions which did not feature text or images, feature engineering sometimes still played an important role in the winning entries. An example of this is the CAUSEEFFECT competition, where the winning entry (described in Appendix A.10) created thousands of features, and then used genetic algorithms to remove non-useful features again.

On the contrary, sometimes the winning solutions are those which go a non-intuitive way and simply use a black-box approach. An example of this is the SOLARENERGY competition (described in Appendix A.6) where the Top-3 entries almost did not use any feature engineering (even though this was the intuitive approach for the competition) – and simply combined the entire dataset into one big table and used a complex black-box model.

2.1.1 Mathematical justification for feature engineering

When using simple models, it is often necessary to engineer new features to capture the right trends in the data. The most common example of this, is attempting to use a linear method to model non-linear behaviour.

To give a simple example of this, assume we want to predict the price of a house H given the dimensions (length l_H and width w_H of the floor plan) of the house. Assume also that the price $p(H)$ can be described as a linear function $p(H) = \alpha a_H + \beta$, where $a_H = l_H \cdot w_H$ is the area. By fitting a linear regression model to the original parameters l_H, w_H, we will not capture the quadratic trend in the data. If we instead construct a new feature $a_H = l_H \cdot w_H$ (the area), for each data sample (house), and fit a linear regression model using this new feature, then we will be able to capture the trend we are looking for.

2.2 Simple models can get you very far

When looking through descriptions of people's solutions after a competition has ended, there is often a surprising number of very simple solutions obtaining good results. What is also surprising, is that the simplest approaches are often described by some of the most prominent competitors.

> "I think beginners sometimes just start to "throw" algorithms at a problem without first getting to know the data. I also think that beginners sometimes also go too-complex-too-soon. There is a view among some people that you are smarter if you create something really complex. I prefer to try out simpler. I "try" to follow Albert Einstein's advice when he said, "Any intelligent fool can make things bigger and more complex. It takes a touch of genius – and a lot of courage – to move in the opposite direction".
>
> (Steve Donoho)

> "My first few submissions are usually just "baseline" submissions of extremely simple models – like "guess the average" or "guess the average segmented by variable X". These are simply to establish what is possible with very simple models. You'd be surprised that you can sometimes come very close to the score of someone doing something very complex by just using a simple model.
>
> (Steve Donoho)

Simplicity can come in multiple forms, both regarding the complexity of the model, but also regarding the pre-processing of the data. In some competitions, regularized regression (as described in Section 4.3) can be the winning model in spite of its simplicity. In other cases, the winning solutions are those who do almost no pre-processing of the data (as seen in for example the SOLARENERGY competition described in Appendix A.6).

2.3 Ensembling is a winning strategy

As described in [Breiman, 2001b], complex models and in particular models which are combinations of many models should perform better when measured on predictive accuracy. This hypothesis can be backed up by looking at the winning solutions for the latest competitions on Kaggle.

If one considers the 10 latest competitions on Kaggle (as described in Appendix A) and look at which models the top participants used, one finds that in 8 of the 10 competitions, model combination and ensembling was a key part of the final submission. The only two competitions where no ensembling was used by the top participants were FACEBOOK and BELKIN, where a possible usage of ensembling was not clear.

> “ [The fact that most winning entries use ensembling] is natural from a competitors perspective, but potentially very hurtful for Kaggle/its clients: a solution consisting of an ensemble of 1000 black box models does not give any insight and will be extremely difficult to reproduce. This will not translate to real business value for the comp organizers. Also I myself think it is more fun to enter competitions where you actually have to think about your model, rather than just combining a bunch of standard ones. In the chess rating, don't overfit, and dark worlds competitions, for example, I used only a single model. (Tim Salimans)

> “ I am a big believer in ensembles. They do improve accuracy. BUT I usually do that as a very last step. I usually try to squeeze all that I can out of creating derived variables and using individual algorithms. After I feel like I have done all that I can on that front, I try out ensembles. (Steve Donoho)

> “ No matter how faithful and well tuned your individual models are, you are likely to improve the accuracy with ensembling. Ensembling works best when the individual models are less correlated. Throwing a multitude of mediocre models into a blender can be counterproductive. Combining a few well constructed models is likely to work better. Having said that, it is also possible to overtune an individual model to the detriment of the overall result. The tricky part is finding the right balance. (Anil Thomas)

Besides the intuitive appeal of averaging models, one can justify ensembling mathematically.

2.3.1 Mathematical justification for ensembling

To justify ensembling mathematically, we will follow the approach of [Tumer and Ghosh, 1996]. We can look at a *one-of-K* classification problem and model the probability of input x

belonging to class i as

$$f_i(x) = p(c_i|x) + \beta_i + \eta_i(x),$$

where $p(c_i|x)$ is an a posteriori probability distribution of the i-th class given input x, where β_i is a bias for the i-th class (which is independent of x) and where $\eta_i(x)$ is the error of the output for class i.

The Bayes optimal decision boundary is the loci of all points $x^\star$ such that $p(c_i|x^\star) = \max_{k \neq i} p(c_k|x)$. Since $f_i(x) = p(c_i|x) + \beta_i + \eta_i(x)$, the obtained decision boundary x_0 (found during model training/fitting) might differ from the Bayes optimal decision boundary $x^\star$ (although x_0 is not the optimal decision boundary, it is still the most likely decision boundary given our training data). We will let $b = x_0 - x^\star$ denote this difference.

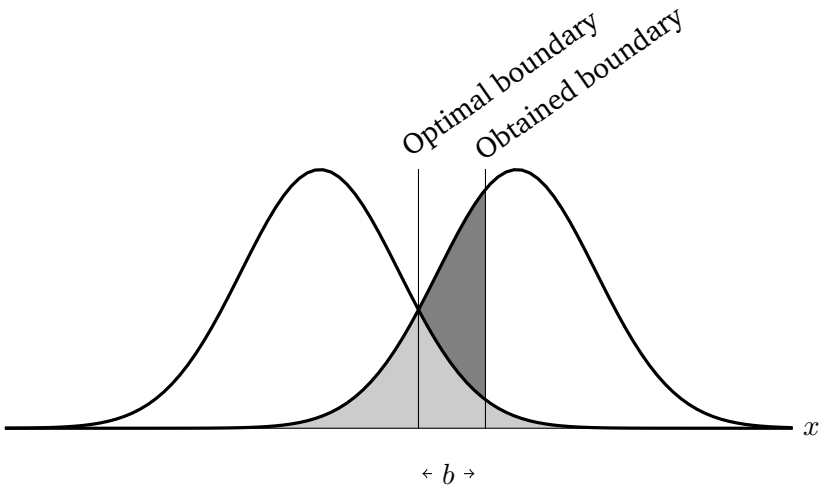


Figure 2.1: Two class distributions together with the optimal class boundary and an example of an obtained boundary. The difference between the boundaries is denoted b. The light shaded area is called the *Bayesian error* E_{Bay} and the dark shaded error is called the *added error* E_{add}.

The goal of ensembling a set of models is to reduce the difference b between the obtained decision boundary and the optimal. Since $x_0 = b + x^\star$ is the obtained decision boundary, we have that:

$$\begin{aligned} f_i(x^\star + b) &= f_j(x^\star + b) \\ p(c_i|x^\star + b) + \beta_i + \eta_i(x^\star + b) &= p(c_j|x^\star + b) + \beta_j + \eta_j(x^\star + b) \\ p(c_i|x^\star + b) + \beta_i + \eta_i(x_0) &= p(c_j|x^\star + b) + \beta_j + \eta_j(x_0). \end{aligned}$$

We now assume that the posterior distributions $p(c_i|x)$ are locally monotonic around the decision boundaries. This assumption is well-founded since the decision boundaries

are typically found in transition regions, where posteriors are not in their local extrema. Using this assumption, we make a linear approximation of $p_k(x)$ around $x^\star$ to get:

$$p(c_k|x^\star + b) \approx p(c_k|x^\star) + bp'(c_k|x^\star),\ \forall k.$$

Using this approximation we get

$$p(c_i|x^\star) + bp'(c_i|x^\star) + \beta_i + \eta_i(x_0) = p(c_j|x^\star) + bp'(c_j|x^\star) + \beta_j + \eta_j(x_0),$$

and since $p(c_i|x^\star) = p(c_j|x^\star)$ we get:

$$\begin{aligned} b(p'(c_j|x^\star) - p'(c_i|x^\star)) &= (\eta_i(x_0) - \eta_j(x_0)) + (\beta_i - \beta_j) \\ b &= \frac{\eta_i(x_0) - \eta_j(x_0)}{s} + \frac{\beta_i - \beta_j}{s}, \end{aligned}$$

where $s = p'(c_j|x^\star) - p'(c_i|x^\star)$ is the difference of the derivatives of the posteriors in the optimal decision boundary. When fitting a model to obtain a decision boundary between the two classes, there is a part of the error which is due to overlap of the classes (the so-called *Bayesian error* E_{Bay}) and a part of the error which is due to our model-fit not being optimal (called the *added error* E_{add}). In Figure 2.1 the dark shaded area is the added error and the light shaded error is the Bayesian error. Since we can not lower the Bayesian error, we can hope to lower the added error by using ensembling.

As given in [Tumer and Ghosh, 1996], when the $\eta_k(x)$ are zero-mean i.i.d. one can show that b is a zero-mean random variable with variance $\sigma_b^2 = \frac{2\sigma_{\eta_k}^2}{s^2}$ and we can write the added error when averaging as

$$E_{\text{add}}^{\text{ave}} = \frac{s}{2}\sigma_{b^{\text{ave}}}^2,$$

and we want to obtain an expression for how this depends on the number of models in the ensemble. Say we fit N models, and let f_i^{ave} be their average, then we have

$$f_i^{\text{ave}} = \frac{1}{N}\sum_{m=1}^{N} f_i^{(m)}(x) = p(c_i|x) + \bar{\beta}_i + \bar{\eta}_i(x),$$

where $\bar{\beta}_i = \frac{1}{N}\sum_{m=1}^{N}\beta_i^{(m)}$ and $\bar{\eta}_i(x) = \frac{1}{N}\sum_{m=1}^{N}\eta_i^{(m)}(x)$. This is due to the fact that the true posterior probability distribution $p(c_i|x)$ is the same for all model fits. Now the variance of $\bar{\eta}_i = \frac{1}{N}\sum_{m=1}^{N}\eta_i^{(m)}$ is given by

$$\begin{aligned} \sigma_{\bar{\eta}_i}^2 &= \frac{1}{N^2}\sum_{l=1}^{N}\sum_{m=1}^{N}\text{cov}\left(\eta_i^{(m)}(x), \eta_i^{(l)}(x)\right) \\ &= \frac{1}{N^2}\sum_{m=1}^{N}\sigma_{\eta_i^{(m)}(x)}^2 + \frac{1}{N^2}\sum_{m=1}^{N}\sum_{l\neq m}\text{cov}\left(\eta_i^{(m)}(x), \eta_i^{(l)}(x)\right), \end{aligned}$$

since we have that $\text{cov}(x,x) = \text{var}(x)$. Using the fact that $\text{cov}(x,y) = \text{corr}(x,y)\sigma_x\sigma_y$ we get that

$$\sigma^2_{\bar{\eta}_i} = \frac{1}{N^2}\sum_{m=1}^{N}\sigma^2_{\eta_i^{(m)}(x)} + \frac{1}{N^2}\sum_{m=1}^{N}\sum_{l\neq m}\text{corr}\left(\eta_i^{(m)}(x),\eta_i^{(l)}(x)\right)\sigma_{\eta_i^{(m)}(x)}\sigma_{\eta_i^{(l)}(x)}.$$

Using the common variance $\sigma^2_{\eta_i(x)}$ which is the average over the m models and the average correlation factor δ_i defined as

$$\delta_i = \frac{1}{N(N-1)}\sum_{m=1}^{N}\sum_{m\neq l}\text{corr}\left(\eta_i^{(m)}(x),\eta_i^{(l)}(x)\right),$$

we can get

$$\sigma^2_{\bar{\eta}_i} = \frac{1}{N}\sigma^2_{\eta_i(x)} + \frac{N-1}{N}\delta_i\sigma^2_{\eta_i(x)}.$$

If we look at the decision boundary offset b of two classes i and j (see Figure 2.1), we get that the variance of b averaged over the N models is given by

$$\sigma^2_{b^{\text{ave}}} = \text{Var}\left(\frac{\bar{\eta}_i(x_0)-\bar{\eta}_j(x_0)}{s} + \frac{\bar{\beta}_i-\bar{\beta}_j}{s}\right) = \text{Var}\left(\frac{\bar{\eta}_i(x_0)-\bar{\eta}_j(x_0)}{s}\right) = \frac{\sigma^2_{\bar{\eta}_i}+\sigma^2_{\bar{\eta}_j}}{s^2}.$$

Consequently we get that

$$\sigma^2_{b^{\text{ave}}} = \frac{\sigma^2_{\eta_i(x)}+\sigma^2_{\eta_j(x)}}{Ns^2} + \frac{N-1}{Ns^2}\left(\delta_i\sigma^2_{\eta_i(x)}+\delta_j\sigma^2_{\eta_j(x)}\right).$$

Now we use that the noise (η) between classes is identically and independently distributed and the fact that

$$\frac{\sigma^2_{\eta_i(x)}+\sigma^2_{\eta_j(x)}}{s^2} = \sigma^2_b,$$

to obtain

$$\begin{aligned}\sigma^2_{b^{\text{ave}}} &= \frac{1}{N}\sigma^2_b + \frac{N-1}{Ns^2}\left(\delta_i\sigma^2_{\eta_i(x)}+\delta_j\sigma^2_{\eta_j(x)}\right)\\ &= \frac{1}{N}\sigma^2_b + \frac{N-1}{N}\frac{2\delta_i\sigma^2_{\eta_i(x)}}{s^2}\frac{\delta_j\sigma^2_{\eta_j}}{2}\\ &= \frac{\sigma^2_b}{N}\left(1+(N-1)\frac{\delta_i+\delta_j}{2}\right).\end{aligned}$$

We now introduce $\delta = \sum_{i=1}^{K}P_i\delta_i$ where P_i is the prior probability of class i. The correlation contribution of each class to the overall correlation is proportional to the class prior probability. Using this and the fact that $E_{\text{add}} = \frac{s}{2}\sigma^2_b$, we get that

$$E^{\text{ave}}_{\text{add}} = \frac{s}{2}\sigma^2_{b^{\text{ave}}} = \frac{s}{2}\sigma^2_b\left(\frac{1+\delta(N-1)}{N}\right) = E_{\text{add}}\left(\frac{1+\delta(N-1)}{N}\right).$$

Looking at the above formula, we observe that a correlation of $\delta = 0$ (i.e. no correlation) will make $E_{\text{add}}^{\text{ave}} = \frac{E_{\text{add}}}{N}$ and that $\delta = 1$ (i.e. total correlation) will make $E_{\text{add}}^{\text{ave}} = E_{\text{add}}$.

The important take-away from these derivations is that ensembling works best if the models we combine have a low correlation. A key thing to note though, is that low correlation between models in itself is not enough to guarantee a lowering of the overall error. Ensembling as described above (there are multiple ways to approach model-ensembling) is effective in lowering the variance of a model (if we only do a single experiment, the variance tells how far off we can expect to be) but not in lowering the bias (difference between expected value and true value, where no bias means that the average prediction will be correct). As an example, say we are averaging a lot of models which all overestimate some value, then their average will still overestimate this value. The concept of bias and variance is visualized in Figure 2.2.

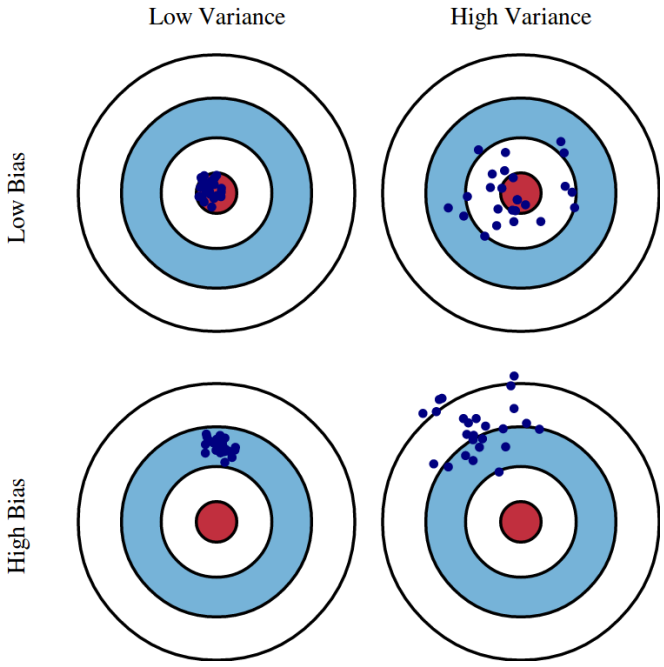


Figure 2.2: A figure visualizing the difference between bias and variance. This figure is taken from `http://scott.fortmann-roe.com/docs/BiasVariance.html`.

2.4 Overfitting to the leaderboard is an issue

During a competition on Kaggle, the participants have the possibility of submitting their solutions (predictions on the public and private test set) to a public leaderboard. By submitting a solution to the leaderboard you get back an evaluation of your model on the public-part of the test set. It is clear that obtaining evaluations from the leaderboard gives you additional information/data, but it also introduces the possibility of overfitting to the leaderboard-scores:

> The leaderboard definitely contains information. Especially when the leaderboard has data from a different time period than the training data (such as with the heritage health prize). You can use this information to do model selection and hyperparameter tuning. (Tim Salimans)

> The public leaderboard is some help, [...] but one needs to be careful to not overfit to it especially on small datasets. Some masters I have talked to pick their final submission based on a weighted average of their leaderboard score and their CV score (weighted by data size). Kaggle makes the dangers of overfit painfully real. There is nothing quite like moving from a good rank on the public leaderboard to a bad rank on the private leaderboard to teach a person to be extra, extra careful to not overfit. (Steve Donoho)

> Having a good cross validation system by and large makes it unnecessary to use feedback from the leaderboard. It also helps to avoid the trap of overfitting to the public leaderboard. (Anil Thomas)

In the 10 last competitions on Kaggle, 2 of them showed extreme cases of overfitting and 4 showed mild cases of overfitting. The two extreme cases were the "Big Data Combine" and "StumbleUpon Evergreen Classification Challenge". In Tables 2.1 and 2.2 the Top-10 submissions on the public test set is shown, together with the results of the same participants on the private test set.

Name	# Public	# Private	Public score	Private score	Entries
Jared Huling	1	283	0.89776	0.87966	65
Yevgeniy	2	7	0.89649	0.88715	63
Attila Balogh	3	231	0.89468	0.88138	137
Abhishek	4	6	0.89447	0.88760	90
Issam Laradji	5	9	0.89378	0.88690	44
Ankush Shah	6	11	0.89377	0.88676	32
Grothendieck	7	50	0.89304	0.88295	73
Thakur Raj Anand	8	247	0.89271	0.88086	37
Manuel Díaz	9	316	0.89230	0.87695	93
Juventino	10	27	0.89220	0.88472	16

Table 2.1: Results of the Top-10 participants on the leaderboard for the competition: "StumbleUpon Evergreen Classification Challenge"

Name	# Public	# Private	Public score	Private score	Entries
Konstantin Sofiyuk	1	378	0.40368	0.43624	33
Ambakhof	2	290	0.40389	0.42748	159
SY	3	2	0.40820	0.42331	162
Giovanni	4	330	0.40861	0.42893	215
asdf	5	369	0.41078	0.43364	80
dynamic24	6	304	0.41085	0.42782	115
Zoey	7	205	0.41220	0.42605	114
GKHI	8	288	0.41225	0.42746	191
Jason Sumpter	9	380	0.41262	0.44014	93
Vikas	10	382	0.41264	0.44276	90

Table 2.2: Results of the Top-10 participants on the leaderboard for the competition: "Big Data Combine"

In the "StumbleUpon Evergreen Classification Challenge" competition, the training data consisted of 7395 samples, and it was generally observed that the data was very noisy[1].

In the "Big Data Combine" competition, the task was to predict the value of stocks multiple hours into the future, which is generally thought to be extremely difficult [2]. The extreme jumps on the leaderboard is most likely due to the sheer difficulty of predicting stocks combined with overfitting.

In the cases where there were small differences between the public leaderboard and the private leaderboard, the discrepancy could also be explained by scores for the top competitors being so close that random noise affected the positions.

Based on empirical evidence, interviews with prominent Kaggler's and on general theory of overfitting, it seems clear that one should think of approaches to avoid overfitting to the leaderboard. Methods to avoid overfitting are described in Section 3.6.

2.5 Predicting the right thing is important

One subject that is sometimes extremely relevant, and other times is not relevant at all, is that of "predicting the right thing". It seems quite trivial to state that it is important to predict the right thing, but it is not always a simple matter.

> "A next step is to ask, "What should I actually be predicting?". This is an important step that is often missed by many – they just throw the raw dependent variable into their favorite algorithm and hope for the best. But sometimes you want to create a derived dependent variable. I'll use the GE

[1] `http://www.kaggle.com/c/stumbleupon/forums/t/6185/what-is-up-with-the-final-leaderboard`
[2] This is what is known as the Efficient Market Hypothesis.

> Flightquest as an example – you don't want to predict the actual time the airplane will land; you want to predict the length of the flight; and maybe the best way to do that is to use that ratio of how long the flight actually was to how long it was originally estimate to be and then multiply that times the original estimate. (Steve Donoho)

There are two ways to address the problem of predicting the right thing: The first way is the one addressed in the quote from Steve Donoho, about predicting the correct derived variable. The other is to train the statistical models using the appropriate loss function. This issue was addressed in the forum of the SOLARENERGY competition:

> "Did anyone achieve a "good" score using RandomForest?
> - Abhishek
>
> I don't think so. Most RF implementations targets MSE, and the metric of this competition is MAE, so i don't think it will give good results.
> - Leustagos (1st place winner)

As an example of why using the wrong loss function might give rise to issues, look at the following simple example: Say you want to fit the simplest possible regression model, namely just an intercept a to the data:

$$x = (0.1,\ \ 0.2,\ \ 0.4,\ \ 0.2,\ \ 0.2,\ \ 0.1,\ \ 0.3,\ \ 0.2,\ \ 0.3,\ \ 0.1,\ \ 100)$$

If we let a_{MSE} denote the a minimizing the mean squared error, and let a_{MAE} denote the a minimizing the mean absolute error, we get the following

$$a_{\text{MSE}} \approx 9.2818, \qquad a_{\text{MAE}} \approx 0.2000$$

If we now compute the MSE and MAE using both estimates of a, we get the following results:

$$\frac{1}{11}\sum_i |x_i - a_{\text{MAE}}| = 9.5909, \qquad \frac{1}{11}\sum_i |x_i - a_{\text{MSE}}| = 16.4942$$

$$\frac{1}{11}\sum_i (x_i - a_{\text{MAE}})^2 = 905.4660, \qquad \frac{1}{11}\sum_i (x_i - a_{\text{MSE}})^2 = 822.9869$$

We see (as expected) that for each loss function (MAE and MSE), the parameter which was fitted to minimize that loss function achieves a lower error. This should come as no surprise, but when the loss functions and statistical methods become very complicated, it is not always as trivial to see if one is actually optimizing the correct thing.

Chapter 3

A Framework for Predictive Modelling

In this chapter we will outline a conceptual framework for approaching predictive modelling competitions – from the initial data exploration and preprocessing to final model selection and combination. For each part of the framework, we will try to describe common methods, tricks and things to consider.

3.1 Overview

In a general setting, the process of predictive modelling in a competitive framework often follows the steps shown here:

1. Explore the data
2. Clean/process the data
3. Construct a statistical model (repeat)
   - Construct features
   - Train model
   - Evaluate model
4. Select best model

> “I learned about the importance of iterating quickly. And trying out many different things and validating, rather than trying to guess the best solution beforehand. (Tim Salimans)

In the following sections we will dive into each of these steps.

3.2 Exploratory analysis

The first step when doing data analysis, should always be to explore and understand the data. Sometimes the data is rather simple and manageable, but sometimes data exploration can be very time consuming.

> “I start by simply familiarizing myself with the data. I plot histograms and scatter plots of the various variables and see how they are correlated with the dependent variable. I sometimes run an algorithm like GBM [gradient boosting machine] or randomForest on all the variables simply to get a ranking of variable importance. I usually start very simple and work my way toward more complex if necessary. (Steve Donoho)

> “Initially If there are continuous variables I would start by getting simply statistics (range, mean, etc) and then plot each one against the dependent variable. If there are categorical variables then histograms. (Robin East)

There is no perfect recipe for how to perform an adequate exploratory analysis, it is a matter of experience and hard work. Still, there are some things which are often a good idea to consider when exploring the data.

3.2.1 Outlier detection

An important reason for exploring the data is to see if there is “wrong” data in the provided dataset. This can be measurements which violates some kind of assumption. In our recent work with data analysis competitions, some of the things which have come up in this category are: Probabilities not summing to one, products which are sent for repair before even being sold and hotel prices of a million dollars per night.

The best way to find outliers, is to either inspect the minimum value, the maximum value and the percentiles of the data numerically, or more commonly to make plots of the different variables.

3.2.2 Plotting the data

One of the best ways to explore data is to visualize it. If one has thousands or maybe even millions of data samples, exploring them manually is not a real possibility. Using the right kinds of visualizations can give the insight into the data which is needed. There is a vast amount of literature and research on how to make good data visualizations, so we will simply mention some of the most common and trivial ones here. We will use a dataset of flowers (the well known iris dataset) for examples.

The most common and simple plot is the scatter plot as shown in Figure 3.1. It shows the relation between two numerical variables using points.

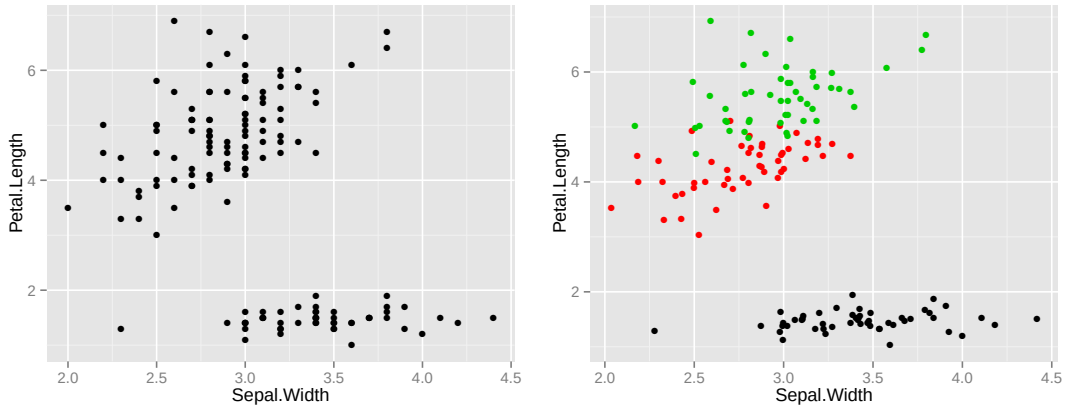


Figure 3.1: Two scatter plots showing the connection between the sepal width and the petal length. In the scatter plot on the right, colors have been added to show the different species of Iris, and a bit of random noise have been added to the points to avoid them being exactly on top of each other.

Using histograms as shown in Figure 3.2, one can visualize a single variable. Values close to each other are combined in bins, and the height of the bins indicate the number of samples binned together.

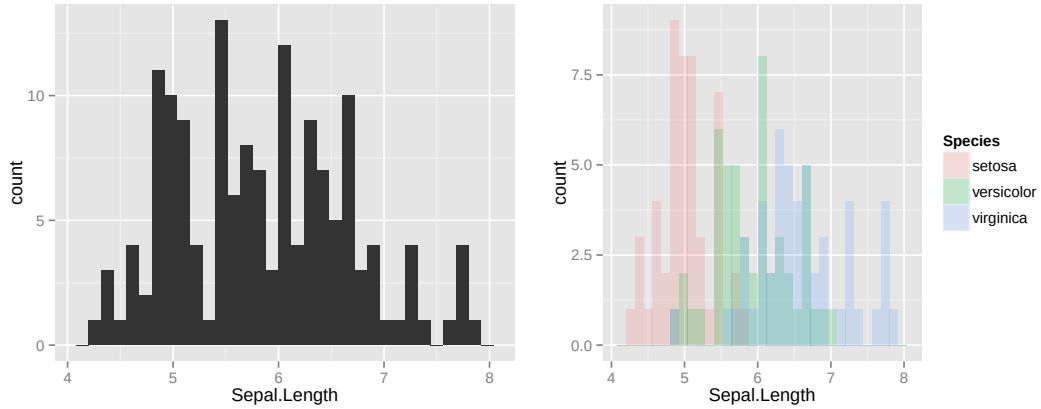


Figure 3.2: Two histograms showing the distribution of sepal length. In the right histogram, the different Iris species are shown with different colors, making it clear that the sepal length differs between species.

Another way to visualize a single variable is to use box-plots as shown in Figure 3.3. Normally, box-plots show the minimum value, maximum value, the lower and upper quartiles and the median in one graphic.

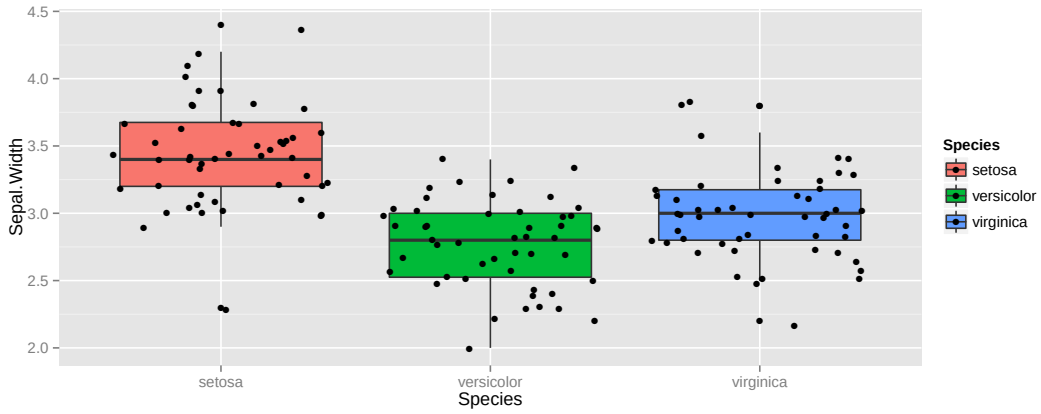


Figure 3.3: A box plot of the sepal width for the three different species. On top of the box plot, the actual data is shown with some horizontal noise added to make all points visible.

3.2.3 Variable correlations

Another thing which is common to explore, is correlations between variables. If there are features which are very correlated, it might be possible to remove some of them to reduce the number of features. If some of the features are correlated with the target values, then those features will have significance when making predictions. A way to visually inspect correlations between features is using correlation matrix-plots as shown in Figure 3.4.

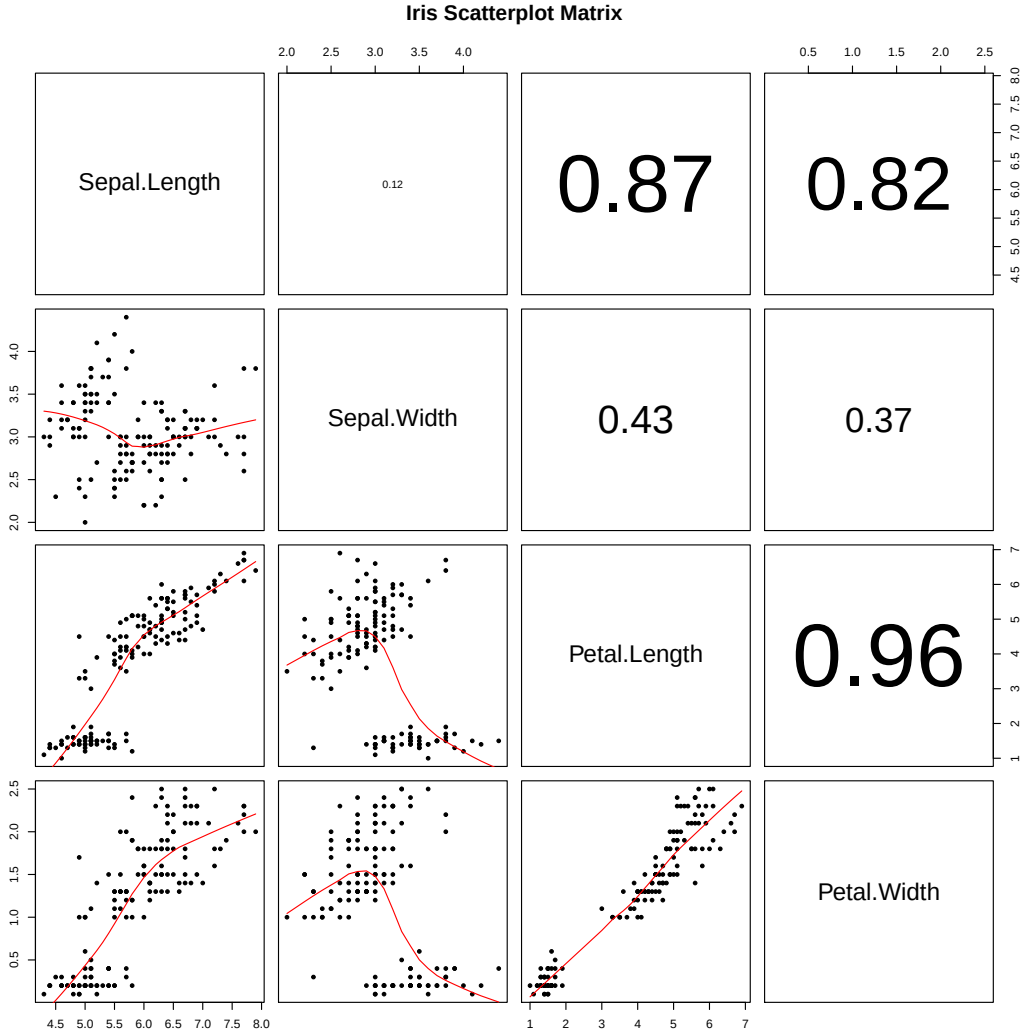


Figure 3.4: A variable correlation matrix-plot. In the diagonal of the matrix, the variable name is stated. In the lower triangular part of the matrix, scatter plots are shown for each pair of variables, including a smoothed curve showing the trend in the data. In the upper triangular part of the matrix, the correlation coefficients are shown, with sizes relative to the amount of correlation.

3.3 Data preparation

It is often stated that around 80%–90% of the time spent on doing data analysis is spent on the process of cleaning and preparing the data [Dasu and Johnson, 2003].

> “Depending on the competition, data preparation can take up a lot of time. For some contests, the data is fairly simple – one flat table – and a user can start applying learning algorithms right away. For other contests – the GE Flightquest contests are an example – there is a lot of data prep work before one can even begin to apply learning algorithms. (Steve Donoho)

One common way to structure data is to use the so-called *tidy-data* format [Wickham, 2012]:

1. Each variable forms a column
2. Each observation forms a row
3. Each table/file stores data about one kind of observation

Other things which should hold for cleaned data are

1. Column names are easy to use and informative
2. Row names are easy to use and informative
3. Obvious mistakes in the data should be removed
4. Variable values are internally consistent
5. Appropriate transformed variables have been added

The paper [Wickham, 2012] describes a consistent way to tidy up data for analysis. According to [Leek, 2013], some of the most important steps to take when doing data preparation is

1. Fix variable names
2. Create new variables
3. Merge datasets
4. Reshape datasets
5. Deal with missing data
6. Take transforms of variables
7. Check on and remove inconsistent values

Since no two datasets are alike, there is not a right way to prepare the data, it is as much a matter of personal preferences as of using the right approach for the specific problem.

3.3.1 Tools for data analysis

There are many tools for doing data analysis, some of them – for example MATLAB and Python – have a very broad focus whereas others like R, are more specialized. Each tool has its own advantages and disadvantages, so it is common for people to employ multiple tools when doing their data analysis.

Here we will attempt to give a short description of the pros and cons of 3 of the main tools[1].

R

One of the most widely used tools for data analysis is R. It is a free software package with a big user base and a large repository of freely available packages implementing most statistical methods and models.

R is based on the S-language and is focused towards more classical statistics. R is very good for doing exploratory data analysis, plotting graphics and handling *dirty* data (combinations of data types, missing values and generally data not in a matrix format).

Currently, the main problem with R is efficiency, both regarding speed but especially memory.

MATLAB

Within the machine learning and applied mathematics community, a very common tool is MATLAB. MATLAB is a non-free (although the open source clone Octave is available) suite of tools for numerical computing.

The strength of MATLAB is in its abilities to effectively handle matrices (MATLAB is an abbreviation of Matrix Laboratory) and its wide range of toolboxes for optimization, image analysis and statistics.

The biggest issues with MATLAB are efficiency, and its lacking focus on handling dirty data – but when the data is structured as simple matrices (or tensors in general), it is highly efficient.

Python

Recently, Python is becoming more popular in the machine learning community. Especially packages such as scikit-learn[2] and pandas[3] have sparked the interest for using

[1] This description will be somewhat biased, but is also heavily based on online debates about the issue.
[2] http://scikit-learn.org/stable/
[3] http://pandas.pydata.org/

	Julia	Python	R	MATLAB
fib	0.91	30.37	411.36	1992.00
parse_int	1.60	13.95	59.40	1463.16
quicksort	1.14	31.98	524.29	101.84
mandel	0.85	14.19	106.97	64.58
pi_sum	1.00	16.33	15.42	1.29
rand_mat_stat	1.66	13.52	10.84	6.61
rand_mat_mul	1.01	3.41	3.98	1.10

Table 3.1: Benchmark times relative to C (smaller is better, C performance = 1.0). These benchmarks and descriptions are given on http://julialang.org/ and include other languages as well.

Python as a tool for data analysis.

One of the main reasons for using Python when doing data analysis, is that Python is built to be an entire programming language by itself – meaning that you can use language features such as object oriented programming to build and structure entire analysis pipelines. Another reason for using Python for data analysis is its efficiency (speed and memory) compared to R and MATLAB.

While the fact that Python is built to be an entire programming language is sometimes an advantage, it is also the main concern with using Python for data analysis. Tools like R are built specifically to be used for data analysis, sometimes making the use of Python for data analysis a less wholesome experience.

Julia

In the future, the modelling language of choice might be Julia[4], promising syntax like MATLAB, speed like C and plotting facilities like R. Table 3.1 shows a comparison of speed for C, R, MATLAB, Python and Julia.

3.3.2 Large datasets

In some competitions the dataset provided is large. When talking about large data here, we mean that the data is too big to fit in memory on a common computer. Either the data in its original form is too large to fit in memory, or the data after necessary pre-processing and feature expansion is too big. Examples of competitions with large datasets are FACEBOOK, SUNNYHASHTAGS, SEECLICKPREDICT, ACCELEROMETER, STUMBLEUPON and BELKIN. Of these competitions, FACEBOOK, SUNNYHASHTAGS, SEECLICKPREDICT and STUMBLEUPON features mainly text-data, where the common feature generation/expansions gives rise to a very large dataset, and BELKIN features a

[4]http://julialang.org/

large dataset of images which by nature are very high-dimensional.

One way to approach very large datasets is the use of special data structures. An example of this is sparse matrices, which are matrices where only non-zero elements are stored. Sparse matrices are very common for natural language processing tasks, where one usually has a huge number of very sparse features. The down-fall of using sparse matrices, is that common operations such as look-up becomes much more time consuming. MATLAB features good implementations of sparse matrices and algorithms on them.

Another way to structure data of large size is using classical relational databases such as MySQL and PostgreSQL. Relational databases stored on disk, scale without problems and features fast look-ups and aggregations. The main issue with using databases is that most of the statistical models will not work out of the box as with data in table format.

Finally, when the data becomes too big, training complex models is simply not feasible. In this case, the most common approach will be to hand-code aggregated features. An example of this is the winning entry of the FACEBOOK competition described in Appendix A.1.2.

Specialized tools for large datasets

In the last years, a lot of research focus has been put on building models, software and conceptual approaches to handle large datasets.

An example of a conceptual approach is the MapReduce framework, with the most popular implementation thereof being Apache Hadoop. MapReduce reduces all computations into two steps, a *map*-step and a *reduce* step. In the *map*-step, the master node divides the input into smaller sub-problems which are then distributed to worker-nodes (this might be done recursively). In the *reduce*-step, the master node collects the answers from worker nodes and combines them. The advantage of formulating computations using the MapReduce framework is that it enables parallel processing of data. Not all problems can be parallelized easily, but when it is possible, a large speed-up is achieved.

An example of software built for large datasets is Vowpal Wabbit[5], self-described as:

> “VW is the essence of speed in machine learning, able to learn from terafeature datasets with ease. Via parallel learning, it can exceed the throughput of any single machine network interface when doing linear learning, a first amongst learning algorithms.

[5] http://hunch.net/~vw/

Vowpal Wabbit is an extremely efficient implementation of different learning algorithms, for example:

- Ordinary Least Squares Regression
- Matrix Factorization
- Neural Networks
- Latent Dirichlet Allocation

and has implementations of different types of regularization and the possibility to handle text-data directly.

3.4 Feature engineering

As discussed in Section 2.1, predictive analysis is all about having the correct features. In some situations, there are no directly useful features (like in image analysis) and in other situations there is a vast amount of irrelevant or correlated features.

In this section we will try to describe some of the common techniques for handling feature-related issues in predictive analysis. We will split the section into two parts, one about reducing the number of features (by removing the least useful features), and one about generating and extracting new useful features.

3.4.1 Feature reduction

A simple and yet very effective way to reduce the number of features, is to manually remove some of them based on intuition about their importance. As an example, say you want to predict the value in dollars for which a product is sold on an online auction (like eBay). Assume that you are given a lot of meta-data about the product, including the time when the item (year, month, date, hour, minute and second) was posted on the website. It is not possible to exclude the possibility that the variables year, month, date and hour could have an important effect on the price (for example, prices might be higher around Christmas time). On the other hand, the second which the product was posted does most likely not hold any information about the sale price. Removing non-useful features will make the statistical models simpler, more robust and less prone to overfitting.

Sometimes the number of features is very large, and it is not always intuitive which features hold predictive power. In this case, we can apply automatic methods for removing non-useful features. Three such methods are *stepwise regression*, *Lasso* (described in Section 4.3) and using advanced models such as *random forests* (described in Section 4.1).

Stepwise regression is regression, where the variables are either introduced gradually, or removed gradually. For simplicity, we will just describe the first type, namely *forward selection*. In forward selection, the initial regression model includes no variables. Then each possible variable is added, and the model fit is evaluated (using some criterion, such as a t-test or the r^2 value). The variable which gives the biggest gain in model fit is put into the model, and the procedure is continued until some criteria is met. There are multiple issues with stepwise regression, the main one being that the same data is being used for model-training and evaluation [Rencher and Pun, 1980].

A more advanced (and very popular) approach to feature reduction is to use for example *random forests* (or other complex methods) for selecting features. For this explanation, we will assume that the reader has the knowledge of random forests presented in Section 4.1. The basic idea behind using random forests to measure variable importance is the following: A random forest is fitted to the data, and the out-of-bag error measure is calculated. To measure the importance of feature j, we permute the values in feature j and compute the out-of-bag error again on this permuted dataset. If the out-of-bag estimate increases a lot by permuting feature j, then that feature must have been important for the predictions. Consequently we rank the features by the difference in out-of-bag error we get by permuting them, normalized in an appropriate way. There are some technicalities regarding categorical variables with different number of categories, which can be solved using so-called *partial permutations* as described in [Deng et al., 2011].

3.4.2 Feature generation and extraction

In many cases, the given features are not directly useful or are very correlated. One example of features that are not directly useful is the one described in Section 2.1.1 where we wanted to predict the price of a house by its width and length (instead of the area). Another example is in image analysis tasks, where the original features (pixel intensities) are not directly useful (in simple models). In other situations, the features are many and highly correlated, making it hard to fit robust statistical models. In this case, one can try to replace the features by a new and smaller set of "better" features, holding the same information as the original features.

Basis expansions

Generating new features as non-linear combinations of the original features is called *basis expansion*. Given a set of features $x_1, x_2, \ldots$, common new features to generate are

$$x_i x_j, \;\; x_i^2, \;\; \log x_i, \;\; \sqrt{x_i}$$

Often, relevant basis expansions can be done manually, as in the house example (see Section 2.1.1) where we create a feature *area* by multiplying features *width* and *length*.

Expansion of categorical features

Often features are not numerical values but categorical, for example which brand a car has or which country a person is from. In this case, it is not meaningful to use this feature directly in for example a linear regression model. One approach to working with categorical features is to *expand* them into multiple boolean features. Say we have a dataset of cars where one feature is the brand of the cars and where the possible values are "Ford", "Audi" and "Volvo":

...	Brand	...
	Ford	
	Ford	
	Volvo	
	Audi	
	Volvo	

Now we can expand this feature into three new features, one for each possible value of the original feature:

...	Ford	Audi	Volvo	...
	1	0	0	
	1	0	0	
	0	0	1	
	0	1	0	
	0	0	1	

With the expanded feature, it is now straightforward to apply for example a linear regression model. The problem with expanding features in this way is that the number of features can expand very rapidly if the categorical features have many categories (although one will often obtain very sparse features).

Text features

In many competitions on Kaggle, the data to be analyzed is text data, either in the form of natural language, webpages or for example tweets from Twitter. When analyzing text data, the task of extracting features is very important since the raw text data is hard to fit into a statistical model out of the box. Common ways to extract features, is to use so-called *bag-of-words*, *n-grams* and *tf-idf*.

The idea behind bag-of-words, is to disregard grammar and order in the text data, but simply storing multiplicity. Say we have the sentence[6]:

> John likes to watch movies. Mary likes too.

[6]Example taken from http://en.wikipedia.org/wiki/Bag-of-words_model on the 2nd of February 2014.

Then we can represent that string as a vector $(1, 2, 1, 1, 1, 1, 1)$ where the 7 features are: *john, likes, to, watch, movies, mary, too*. Note that we have a 2 since the word "likes" is used twice in the sentence. By converting a dataset of text using bag-of-words, one gets a data-matrix with a sample per document, and a feature for each unique word in the entire dataset (consequently giving a very high-dimensional – yet often very sparse – dataset).

Sometimes the context is lost when only considering words by themselves. A common approach to handling this issue is using n-grams which is a generalization of bag-of-words. An n-gram is a vector of n consecutive words in a text. For example, the seven 2-grams (also called bi-grams) in the above text are:

$$\{\text{john, likes}\},\ \{\text{likes, to}\},\ \{\text{to, watch}\},\ \{\text{watch, movies}\}$$

$$\{\text{movies, mary}\},\ \{\text{mary, likes}\},\ \{\text{likes, too}\}$$

and now these bi-grams can be used as features as well.

Finally, another very common way of handling text data, is to use the so-called *term frequency - inverse document frequency* (tf-idf) weighting scheme. The basic idea in tf-idf is to down-weight words which are very common in the dataset. Say you want to find documents which are relevant to the query "the cheapest pizza". A simple approach is to find all documents containing the words "the", "cheapest" and "pizza". Since there are most likely multiple documents containing those words, we need to order them by relevance. One possible way to do that, is to count the number of times each word occur in the document (the term frequency), giving priority to the documents using the words a lot. The problem with this approach is that the word "the" is so common, that it will most likely overshadow "cheapest" and "pizza". A solution is to introduce an inverse document frequency factor, reducing the weight of terms which are very common in the dataset. Mathematically, the tf-idf score of a term t in a document can be computed as the product $tf(t) \cdot idf(t)$ where:

$$tf(t) = \frac{\text{Number of occurences of term } t \text{ in the document}}{\text{Total number of terms in document}}$$

$$idf(t) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents with } t \text{ in it}}\right).$$

Non-negative matrix factorization

When the data is on matrix-form, one can employ Non-negative Matrix Factorization (NMF) to extract a new (smaller) set of features without loosing too much information. The idea is to factorize the data matrix X into factors W, H such that H contains a new set of features and W expresses the original samples in these new features. Let X be an $n \times m$ matrix, where the n rows corresponds to samples of m features each. Now the objective is to find W, H such that:

$$X \approx WH, \qquad W \geq 0, H \geq 0.$$

where $W \in \mathbb{R}^{n \times k}$ and $H \in \mathbb{R}^{k \times m}$. We obtain such a decomposition using the following multiplicative update rules derived in [Lee and Seung, 2000]:

$$W_{i,d} \leftarrow W_{i,d} \frac{\sum_j W_{i,d} \frac{X_{i,j}}{(WH)_{i,j}}}{\sum_j H_{d,j}}, \qquad H_{d,j} \leftarrow H_{d,j} \frac{\sum_i W_{i,d} \frac{X_{i,j}}{(WH)_{i,j}}}{\sum_j W_{i,d}}$$

The reason for using the non-negativity constraint $W, H \geq 0$ is to avoid having features "cancelling each other out". It also makes the features extracted easier to interpret. As an example, Figure 3.5 shows a set of features extracted from a dataset of images of human faces.

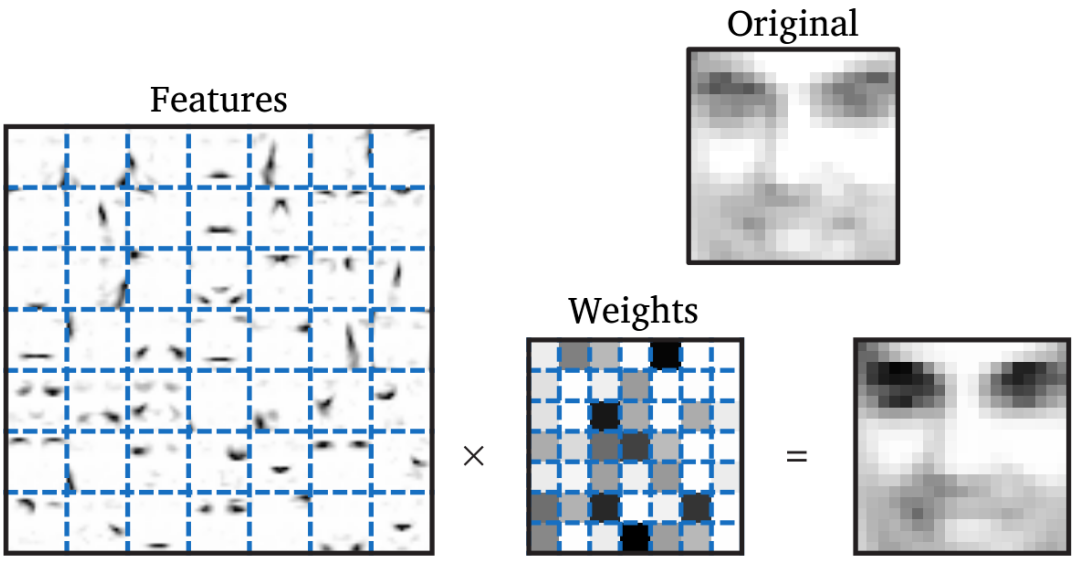


Figure 3.5: An example of features extracted using NMF taken from [Lee and Seung, 1999] when trained on a dataset of images of faces.

Besides using the non-negativity constraint, it is also possible to introduce other constraints on the decomposition. One common possibility is to introduce a sparsity constraint (in the same way as Lasso introduces sparsity in regression models) on the features [Hoyer, 2002].

Principal component analysis

Another popular method for reducing dimensionality of the data is Principal Component Analysis (PCA). Like with NMF, the motivation is to find a new smaller set of features which is able to explain the data. Principal Component Analysis (PCA) is an orthogonal transformation of the data such that we achieve maximal variance along the new axes (features). The data matrix X is transformed as

$$S = XL,$$

where S is called the *scores* and L is called the *loadings* (weights). S is $n \times k$ where $k \leq m$, and represents the coordinates of the data in the new orthogonal basis. The columns of S are the principal components which are linearly uncorrelated. L is a $m \times k$ rotation matrix (an orthogonal matrix) where the columns are of unit length. PCA can be done using eigen-analysis of the covariance matrix $X^T X$ or using Singular Value Decomposition of the data matrix. For dimensionality reduction, one uses S as the new data matrix instead of X.

3.5 Model training

When we have cleaned the data and constructed the features we need, then the next logical step is to select and train an appropriate model. There is no simple algorithm for selecting the best model for a task, but there are some general guidelines which can be outlined. When a model has been selected, there might be some issues with the data which needs to be adressed before training, for example an *imbalanced* dataset, or a large amount of missing data. Finally, some models require a set of hyper-parameters which we need to choose in some systematic way.

3.5.1 Selecting a model

Overall, every prediction task is one of two categories: Regression or classification. In regression problems, we want to predict numerical outcomes and in classification problems, we want to predict discrete outcomes (or classes). Very rarely does it make sense to use a regression model on a classification problem or vice versa. There is no magic formula for choosing the right model, but we will list some of the most common models for regression and classification here:

Regression

- Linear Regression

  Including different types of regularization
- K nearest neighbors
- **Regression Trees**
- Ensembling Methods

  Random Forests

  Boosting
- **Neural Networks**

Classification

- **Logistic Regression**
  - **Including different types of regularization**
- Support Vector Machine
- K nearest neighbors
- Naïve Bayes
- **Decision Trees**
- Ensembling Methods
  - **Random Forests**
  - Boosting
- **Neural Networks**

where the ones marked with bold are described in Chapter 4. All of the methods listed above are described in for example [Bishop, 2006].

3.5.2 Imbalanced data

In some cases, the training data is not balanced in the same way as the test data (different ratio between positive and negative samples), which might lead to a wrong model fit. Consider for example the case where we are trying to predict if tumor is malignant or not. If the training data consists of 10 malignant tumors and 9990 non-malignant tumors, any model will be able to get an accuracy of 0.1% by simply classifying any tumor as non-malignant. In this case, it will rarely be the case that the model learns anything useful except the prevalence of the classes in the data. If the test data then contains 50/50 malignant and non-malignant tumors, the model always predicting non-malignant will perform very badly.

There are multiple ways to handle the problem of imbalanced classes. The two most common are under-sampling and over-sampling.

Under-sampling refers to reducing the amount of samples in the class which is over-represented. This can be done by simply removing samples at random or by selecting a subset of appropriate samples.

Over-sampling is the opposite approach, namely expanding the samples from the under-represented class. This can be done by replication of samples or by intelligently combining samples.

Finally it is also possible to handle the situation without directly over- or under-sampling the classes. If the objective during model training is to minimize some loss-function which is computed over the samples, then one can introduce weights on the samples. A weight of 2 on a sample, corresponds to over-sampling that specific sample by duplication.

3.5.3 Imputation of missing data

A common problem when working with real world datasets, is the case of missing data. If the number of samples with missing data is very small, it might not impact the result very much to remove them, but if a large proportion of samples have missing data, then it is necessary to have a way of handling it.

A common way to handle missing data, is to attempt to infer the missing values using the rest of the dataset – called imputation. One of the simplest ways to impute missing data, is to compute some summary statistic (often the mean or the median) for each feature (using only the non-missing values), and then impute that value at the missing values. For example, given the following values for a specific feature:

$$10,\ 15,\ ?,\ ?,\ 5,\ 10,\ ?,\ 5,\ 10$$

we could compute the median to be 10, and then impute 10 instead of the missing values to obtain

$$10,\ 15,\ 10,\ 10,\ 5,\ 10,\ 10,\ 5,\ 10$$

Another way of imputing missing values includes building simple regression/classification models using other features. In this way, we can give a better estimate of the missing values using non-missing values in the other features. Yet another common method of missing data imputation is to use k-nearest neighbors, where one finds the k samples which are closest by some metric, and impute their values.

3.5.4 Selecting hyper parameters

Many statistical models require the tuning of hyper parameters – for example the structure of neural networks, the λ-parameters in elastic net and the structural parameters for decision trees. Finding appropriate values for these hyper parameters is often done by manual selection, but a more systematic way is to try a set of different values and choose the ones which gives the best validation scores (see Section 3.6 for a description of how to measure validation errors).

3.6 Evaluation

3.6.1 Cross validation

When given a dataset with the true target values, we can train a statistical model. Assume that the dataset consists of two components: An input-matrix X and an output-

vector y, where X is an $n \times m$ matrix and y is an $n \times 1$ vector. The input-matrix X has one row per sample consisting of m features, and the y vector contains the true output for each sample.

Now the task is to fit a statistical model p, which given the i-th sample (the i-th row) from X, can predict the i-th element in y. We want p to be fitted such, that if we get a new input-matrix X' but do not get to observe the corresponding outputs y', we can still make accurate predictions about y' (that is, we want p to generalize to new input). Cross validation is a method for evaluating how good a model fit is able to generalize to new input using only the given data X, y.

In k-fold cross-validation, the original dataset (X, y) is randomly partitioned into k parts of approximately equal size. Of these k parts, a single part is retained as validation data for testing the model (using the appropriate metric), and the remaining $k-1$ parts are used as training data. We repeat this for each of the k partitionings (folds), and average the k results together. The idea is that for each fold, we train and evaluate the model on disjoint data.

Normally k is set to 10, but this is a parameter which can be varied. In the extreme case where $k = n$ this method is known as leave-one-out cross validation since we only leave a single sample out in each of the n folds. When model training is time consuming and the datasets are large, using a large k becomes infeasible.

3.6.2 Validation on temporal data

In some cases it is not straight forward to use cross validation. One such example is predictions of time series. Assume that we are given a series of measurements $t_1, \ldots, t_n$ over n time steps, for examples measurements of the temperature in Copenhagen. Now we want to build a model to predict the temperature in the future $t_{n+1}, t_{n+2}, \ldots$.

If we want to validate that our predictive model generalizes, we could be tempted to use cross validation on the data $t_1, \ldots, t_n$, but this will not work. If we split our data such that we have t_i and t_{i+2} in the training data, and need to predict t_{i+1}, then we will most likely make a very good estimate (since we know the temperature both before and after). We might also end in a situation where we need to predict backwards in time, which might need a different modelling approach.

The most common solution to validating time series models for generalizability, is to split the dataset into two parts $t_1, \ldots, t_i$ and $t_{i+1}, \ldots, t_n$, train the model on $t_1, \ldots, t_i$ and predict on $t_{i+1}, \ldots, t_n$ for validation. If the task is to predict T time-steps into the future, then the data split can be made as $t_1, \ldots, t_i$ and $t_{i+T}, \ldots, t_n$, which will test the models ability to predict T and more time-steps into the future.

> “Cross-validation is very important. Once I have a good feel for the data, I set up a framework for testing. Sometimes it is cross validation. If it is time-related data where you are using earlier data to predict later data (SeeClickFix is a good example), it is better to split your training data into “earlier” and “later” and use the earlier to predict the later. This is more realistic compared to the actually testing. (Steve Donoho)

3.6.3 Evaluation measure

When evaluating the model, it is important to use the correct evaluation measure. In predictive modelling competitions, this is naturally the measure used for determining the winning solution. See Section 2.5 for a more in depth discussion of the importance of using the correct evaluation measure.

3.6.4 Leaderboard

In the competition setting on Kaggle, the public leaderboard gives an additional possibility for evaluating models. After doing predictions on the given test set, these predictions can be submitted to Kaggle, which will then evaluate them on the public part of the test set (for additional information about the Kaggle setup, see Section 1.2) and put the result on the leaderboard. This setup enables users to determine if they are overfitting their models locally, since this will result in a lower score on the leaderboard than expected from local evaluations. It is important to be cautious when using the leaderboard, as described in Section 2.4.

3.6.5 Reexamining most difficult samples

When doing predictions on a local validation set, it is possible to determine on which data samples the model made the most erroneous predictions. Using this information, it is possible to reexamine these samples and either change the predictive model or generate new features enabling a more accurate prediction on these samples.

3.7 Model selection and model combining

The final step in the process of doing competitive data analysis, is to select the model you want to submit to the competition. Using local validation scores, and public leaderboard scores gives usable information about which model will perform best on the private test set.

Sometimes it is possible to select multiple models for evaluation on the final leaderboard. In this case the two most obvious strategies is to either select the best performing models, or to select models which are very different in structure or complexity. The last strategy is especially relevant if the chance of overfitting is large, where having a simple

back-up model can reduce the risk of dropping many positions on the leaderboard.

Finally, if one has trained multiple models, it is natural to think about ensembling these models together as described in Section 2.3. A way to combine models for the final submission is to weight them by their validation and leaderboard scores such that the most promising models will get a higher priority in the final ensemble. It is also possible to train a model for combining the outputs of other models (known as model stacking), for example having a logistic regression model for combining multiple complex models such as neural networks.

Chapter 4

Theory

In this chapter, we will describe the underlying mathematical theory behind the models used throughout this thesis. We will describe decision trees, random forests, LambdaMART and logistic regression which were used for the Expedia challenge discussed in Chapter 5, and we will describe convolutional neural networks which were used for the GalaxyZoo challenge discussed in Chapter 6.

The theory described in this chapter will not focus on deriving everything from scratch, but some derivations will still be provided for completeness. The focus is on providing some details about the methods to give a better understanding of why and how they work.

4.1 Random forests

One of the most successful models in modern data science competitions is random forests [Breiman, 2001a, Ho, 1998, Amit and Y, 1997, Ho, 1995]. The idea behind random forests is to build a large ensemble of so-called *decision trees*, constructed in such a way that their predictions have low correlation. By building de-correlated models, random forests take advantage of the possibilities from ensembling, consequently lowering the overall variance of the model.

4.1.1 Decision trees

The base-model in random forests is a decision tree. A decision tree is a model with high expressive power, which in some ways model human decision making and reasoning. Usually, a decision tree is a binary tree, where each node represents a question about a variable – eg. is $x_1 \leq 0$ – and where the two outgoing edges from that node corresponds to the two possible answers to the question. For classification problems, each leaf-node in the bottom of the tree corresponds to a possible outcome-class. As an example, Figure 4.1 shows a decision tree for a classification problem with 3 variables x_1, x_2, x_3 and two output-classes A and B.

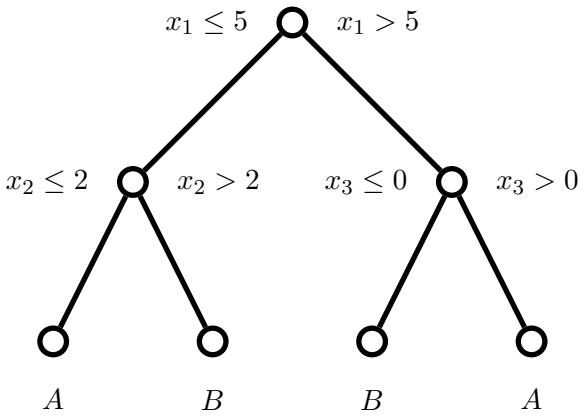


Figure 4.1: Decision tree

When building/training decision trees one usually grows large trees such that all training data is classified correctly (thus overfitting the tree to the training data), and then prune away some of the tree again to avoid overfitting. When using decision trees in random forests, the trees are normally not pruned, consequently giving overfitted trees with high variance.

When building a decision tree for a regression problem (sometimes called a regression tree), one can either have a constant numerical prediction in each leaf-node or build simple individual regression models for the samples in each leaf-node.

When constructing a decision tree from data, one starts with all data in the top node, decides on a split criterion, splits the data according to this criterion and continues this recursively until the number of samples in each node reaches some small chosen constant. For deciding on the split criterion, one looks at all variables, and for each tests all possible splits (taking time $O(n \log n)$ if done cleverly, where n is the number of samples). For each possible combination of variable and split, one evaluates a measure of how good the split is. One such measure is the Gini-index. For a tree T, we define the Gini-index $G(T)$ as

$$G(T) = 1 - \sum_{j=1}^{k} \left(\frac{\sum_{i=1}^{N_T} I(y_i = k)}{N_T} \right)^2,$$

and for a split into subtrees T_1, T_2, we define the Gini-index $G(T_1, T_2)$ as

$$G(T_1, T_2) = \frac{N_{T_1}}{N_{T_1} + N_{T_2}} G(T_1) + \frac{N_{T_2}}{N_{T_1} + N_{T_2}} G(T_2).$$

We see that $G(T)$ is small when the classes are skewed (as visualized in Figure 4.2), for example having only samples from a single class. The split with the lowest Gini-index is chosen.

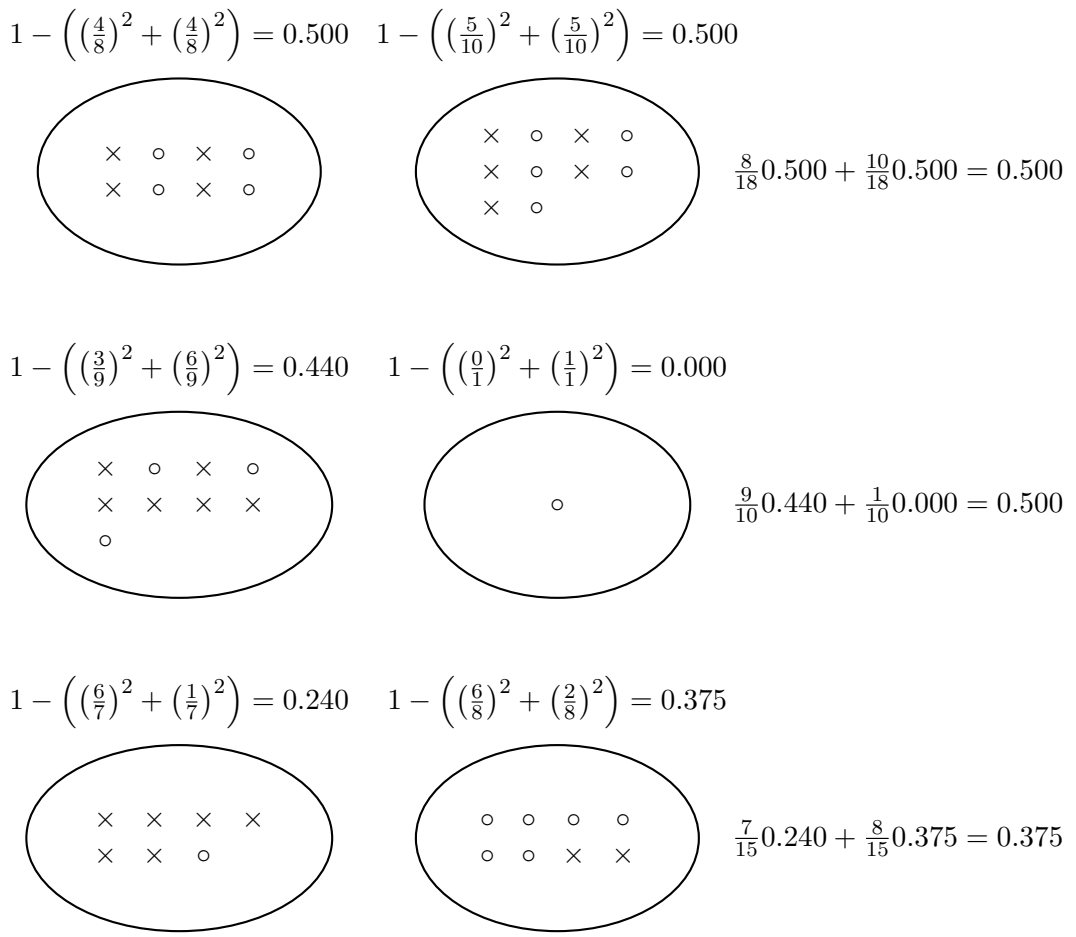


Figure 4.2: An illustration of how the Gini-index is calculated. The Gini-index favors (by assigning a low score) splits where the classes are separated.

4.1.2 Random forest

When training a random forest of k trees for a dataset with N sample and M variables, we do the following to train each of the k trees: We make a bootstrap sample of size n from the N data-samples and use the rest of the data for making an error-estimate (the so-called out-of-bag estimate). For each node in the tree we randomly choose m of the total M variables, and between these find the optimal split.

When making predictions with a random forest on some new sample, we feed it through all k trees, and combine the outputs using some function (often used functions are majority vote for classification and average for regression).

One of the biggest issues for random forests is when a lot of features are correlated, since this can result in nodes which have no reasonable variables to split on.

4.2 RankNet, LambdaRank and LambdaMART

LambdaMART is an approach to training an advanced model for ranking items which has been successful. For example an ensemble of LambdaMART models won the first place in the 2010 Yahoo! Learning To Rank Challenge [Chapelle et al., 2011]. LambdaMART is a so-called *boosted* tree version of LambdaRank which in turn is based on RankNet. These models are introduced in various places, but is collected in [Burges, 2010]. We will describe the models like they are described in [Burges, 2010].

4.2.1 Learning to rank

One usually categorises supervised machine learning problems into the two categories: classification and regression. Another problem to consider is that of ranking lists of items, the so-called *learning to rank* (LTR) problem.

In the LTR-problem we construct a ranking model from training data. The training data is lists of items which have a partial order. Often the items in the lists are also denoted with a numerical score or a binary relevance-judgement. We want to build a model from the traning data, which given a new list, will rank/order/permute them in a way similar to the training data.

One of the most well-known examples of learning to rank is Google's algorithms for ranking search results for a query.

There are generally three types of features in LTR-problems: Static, dynamic and query-specific. The static features are those which does not depend on the query but only on the item (e.g. the number of words on the webpage), the dynamic features are those that depend on both the query and the item (e.g. the number of words in common between the query and the webpage) and the query specific features only depends on the query (e.g. the number of special characters in the query string).

There are multiple metrics for evaluating ranking algorithms, the two most well-known being Mean Average Precision (MAP) and DCG/NDCG (Discounted Cumulative Gain and Normalized Discounted Cumulative Gain). Other than these two, commonly used metrics are Mean Reciprocal Rank and Kendall's Tau, and recently the metrics Expected Reciprocal Rank and Yandex's pfound have been proposed.

Approaches to learning to rank

The approaches to LTR-problems can be divided into three categories, the *pointwise approaches*, the *pairwise approaches* and the *listwise approaches*.

The pointwise approaches assume that each query-item pair has a corresponding numerical relevance score. Then the problem of ranking a list of items is reduced to the regression problem of predicting the score of each query-item pair and then sorting by

those predictions.

The pairwise approaches looks at pair of items within a query, and tries to classify which of the items is most relevant to the query. One then minimizes the expected number of inverted pairs in the query.

The listwise approaches are tied more directly to the metrics mentioned above. Here the idea is to directly optimize the metrics over all queries in the training data. The difficulty here is that the metrics are rarely continuous in the parameters of the ranking models, and consequently the optimization is done approximately.

4.2.2 RankNet

Say we are ranking items $U_1, \ldots, U_n$ and that the item U_i has a corresponding feature vector x_i and that we have a model $f : \mathbb{R}^m \to \mathbb{R}$ where m is the number of variables. We require for f that it is a differentiable function of its parameters $w_1, \ldots, w_m$.

When training, we partition the data by queries. Within a query, we present each pair of items U_i and U_j where $i \neq j$ to the model. The model computes the scores for each item $s_i = f(x_i)$ and $s_j = f(x_j)$. We let $U_i \triangleright U_j$ denote the event that U_i should be ranked higher than U_j. Now we map the two outputs of the model to a learned probability that U_i should be ranked higher than U_j using a sigmoid function

$$P_{ij} \equiv P(U_i \triangleright U_j) \equiv \frac{1}{1 + e^{-\sigma(s_i - s_j)}},$$

where σ is a shape parameter for the sigmoid. We can then apply the cross entropy cost function penalizing the deviation of the model output from the desired probabilities. Let $\bar{P_{ij}}$ be the known probability that item U_i should be ranked higher than U_j. Then the cost is given by

$$C = -\bar{P_{ij}} \log P_{ij} - (1 - \bar{P_{ij}}) \log(1 - P_{ij}).$$

For a given query, let $S_{ij} \in \{0, 1, -1\}$ be defined as 1 if item i has been labeled more relevant than item j, -1 if opposite and 0 if they have the same label. We will assume that the desired ranking is known so that $\bar{P_{ij}} = \frac{1}{2}(1 + S_{ij})$. Combining the equations above gives us

$$C = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)}).$$

Now we have that

$$\frac{\partial C}{\partial s_i} = \sigma \left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}} \right) = -\frac{\partial C}{\partial s_j}.$$

Using this gradient, we can update the weights and consequently reduce the cost using stochastic gradient descent:

$$w_k \to w_k - \eta \frac{\partial C}{\partial w_k} = w_k - \eta \left(\frac{\partial C}{\partial s_i} \frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j} \frac{\partial s_j}{\partial w_k} \right),$$

where η is a positive learning rate and where we have the convention that if two quantities appear as a product and share an index, then that index is summed over, for example

$$\frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} = \sum_i \left(\frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k}\right).$$

We underline the idea of gradient descent which is to update the model using the gradients of the cost function with respect to the weights – which is the core of RankNet, LambdaRank and LambdaMART. For ranking problems the cost does not always have well-posed gradients and sometimes the model does not even have differentiable parameters. In these cases we will bypass the computation of $\partial C/\partial w_k$ by directly modelling $\partial C/\partial s_i$.

This idea can be approached using the following factorization

$$\begin{aligned}\frac{\partial C}{\partial w_k} &= \frac{\partial C}{\partial s_i}\frac{\partial s_i}{\partial w_k} + \frac{\partial C}{\partial s_j}\frac{\partial s_j}{\partial w_k} \\ &= \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}}\right)\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right) \\ &= \lambda_{ij}\left(\frac{\partial s_i}{\partial w_k} - \frac{\partial s_j}{\partial w_k}\right),\end{aligned}$$

where we define

$$\lambda_{ij} \equiv \frac{\partial C}{\partial s_i} = \sigma\left(\frac{1}{2}(1 - S_{ij}) - \frac{1}{1 + e^{\sigma(s_i - s_j)}}\right).$$

Let I denote the set of pairs of indices (i, j) for which we desire U_i to be ranked differently from U_j (for a given query). We chose I such that each pair is included just once, so we only look at pairs (i, j) where we want to rank U_i higher than U_j.

If we now look at the total contribution to the weight update of w_k (the sum over the pairs in I) we get

$$\delta w_k = -\eta \sum_{\{i,j\}\in I}\left(\lambda_{ij}\frac{\partial s_i}{\partial w_k} - \lambda_{ij}\frac{\partial s_j}{\partial w_k}\right) \equiv -\eta \sum_i \lambda_i \frac{\partial s_i}{\partial w_k},$$

where λ_i is λ_{ij} summed over j. To compute λ_i, we use

$$\lambda_i = \sum_{j:\{i,j\}\in I} \lambda_{ij} - \sum_{j:\{j,i\}\in I} \lambda_{ij}.$$

One can think of the λ's as forces acting on the items, pulling them up or down in the ranking. The λ's are accumulated for each item, and then the update is done.

4.2.3 Normalized discounted cumulative gain

One of the more common metrics for evaluating rankings, is the Normalized Discounted Cumulative Gain (NDCG). NDCG is a normalized version of the Discounted Cumulative Gain (DCG). DCG@k for a query is defined as

$$\text{DCG}_k = \sum_{i=1}^{k} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)},$$

where k is the truncation level and rel_i is the "relevance" of the item we choose to rank as number i. The truncation level k is the maximum number of results in the query to consider for evaluating the score.

Now we can define NDCG@k as

$$\text{NDCG}_k = \frac{\text{DCG}_k}{\text{IDCG}_k},$$

where IDCG@k is the ideal DCG (maximum possible DCG for this query). From the definition, we see that NDCG is a score between 0 and 1.

The problem with NDCG (and other measures used for ranking problems) is that viewed as a function of the model scores (s_i), it is everywhere either discontinuous or flat, making gradient descent problematic since the gradient is always zero or not defined [Burges, 2010].

4.2.4 LambdaRank

One of the ideas behind LambdaRank [Burges, 2010] is to directly write the gradients (and consequently the update rules) without deriving them from a cost-function. We imagine that there is a utility C (we are now maximizing NDCG, so we will use utility instead of cost) such that:

$$\lambda_{ij} = \frac{\partial C}{\partial s_i} = \frac{-\sigma}{1 + e^{\sigma(s_i - s_j)}} |\Delta_{\text{NDCG}}|,$$

and since we are now maximizing C, we alter the update rule to be:

$$w_k \to w_k + \eta \frac{\partial C}{\partial w_k},$$

giving

$$\delta C = \frac{\partial C}{\partial w_k} \delta w_k = \eta \left(\frac{\partial C}{\partial w_k} \right)^2 \geq 0.$$

This solves the problem that although NDCG viewed as a function of model scores is flat or discontinuous, we can maximize it by computing the gradients after the items have been sorted by their scores. In [Burges, 2010], it is empirically shown that the

gradients stated above actually optimize NDCG. This is done by looking at the obtained weights $w^\star$, and then estimating a smoothed version of the gradients with respect to each weight w_i individually. Now they check if the average NDCG-score over all queries is a maximum for each w_i indicating that it is indeed a maximum or a saddle point. To test for a saddle point, they do a one-sided Monte Carlo test by uniformly sampling sufficiently many random directions in weight space, and for each checking if moving in that direction will decrease the average NDCG-score.

4.2.5 LambdaMART

Multivariate Adaptive Regression Trees (MART) is also known as Gradient Tree Boosting and was introduced in [Friedman, 2000]. MART is a class of algorithms rather than a single algorithm which can be trained to minimize general cost functions.

Let $h_m(x)$ be a decision tree with J leaves. Then $h_m(x)$ partitions the input space into J distinct regions $R_{1m}, \ldots, R_{Jm}$ where each region corresponds to a constant prediction. We can write $h_m(x)$ as:

$$h_m(x) = \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm}),$$

where γ_{jm} is the prediction in R_{jm}. If we start with a model $F_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$, we can now update the model $F_m(x)$ as:

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J} \gamma_{jm} I(x \in R_{jm}),$$

where

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)).$$

Conceptually, this can be seen as doing gradient descent in function space, minimizing a loss function L as a function of the model scores evaluated at the training samples $f(x_1), \ldots, f(x_n)$. Each gradient descent step is taken from a restricted class of functions (the base learners) – in this case regression trees.

Since MART optimizes using steepest descent and LambdaRank works by specifying derivaties of the cost-functions at any point during training with respect to the model scores, the two models are well suited for being combined giving LambdaMART. LambdaMART is basically just gradient tree boosting with the derivatives of specified by LambdaRank. A few more details about how to take the appropriate step sizes in the gradient descent are given in [Burges, 2010].

4.3 Logistic regression with regularization

Binomial logistic regression can be used to classify observations into two groups. The idea is to model the probability of a success using a non-linear function in the input x.

The model can be written on the form

$$\log \frac{\Pr(\text{Success}|x)}{1 - \Pr(\text{Success}|x)} = \beta_0 + \beta_1 x.$$

Solving this for $\Pr(\text{Success}|x)$ gives us

$$\Pr(\text{Success}|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}},$$

which is called the logistic function and will from here be denoted $f(x)$. The output of $f(x)$ is in the range $[0; 1]$ (for any real input x) and can be interpreted as a probability of success. Compared to linear regression, there is no closed-form expression for the coefficients maximizing the likelihood function. Consequently, numerical methods are used to train logistic regression classifiers, that is to find coefficients β_0, β_1 maximizing the likelihood:

$$\ell(\beta) = \sum_{i=1}^{N} \{y_i(\beta_0 + \beta_1 x) - \log(1 + e^{\beta_0 + \beta_1 x})\}.$$

4.3.1 Ridge regression

A common problem with logistic and linear regression is that highly correlated variables can result in large coefficients which cancel each other out, thus leading to a model with high variance. One way to overcome this problem is to impose a prior assumption on the size of the coefficients in the model - this is known as regularization. The most widely used method of regularization is ridge regression (or Tikhonov regularization), which changes the usual problem of maximizing the likelihood function $\ell(\beta)$ to maximizing $\ell(\beta) - \lambda \|\beta\|^2$. Imposing this additional penalty on the norm of the parameters will shrink them towards zero. Another feature of ridge regression is that it allows for having more features than data samples which is not normally possible due to the problem being ill-conditioned.

Changing the prior on the parameters slightly by subtracting $\lambda \|\beta\|$ (the L_1 norm) instead of $\lambda \|\beta\|^2$ changes the solutions by causing small parameters to become 0. Regularizing using the L_1-norm is known as the Lasso-method (or basis pursuit). A useful result from using L_1 regularization is that features which do not add value to the model are dropped completely, consequently reducing variance in the solution.

4.3.2 Elastic net

One can also combine L_1 (Lasso) and L_2 (ridge regression) penalties for regularizing:

$$\hat{\beta} = \arg\max_{\beta} \left(\ell(\beta) - \lambda_2 \|\beta\|^2 - \lambda_1 \|\beta\|_1 \right).$$

This is known as elastic net regularization. Elastic net incorporates the advantages of both Lasso and ridge regression, with the disadvantage of introducing one additional hyper parameter to fit [Hastie et al., 2001].

4.4 Neural networks and convolutional neural networks

An Artificial Neural Network - more commonly called a Neural Network - is a tool for modelling complex relations between input and output. We will use the notation given in [Bishop, 1995] to describe neural networks. Once the relations between input and output are known, a neural network can behave as a classifier for predicting the output of new input. A neural network is a graphical model, where neurons are connected by synapses in an attempt to mimic the cognitive structure of the brain. Neural networks consist of layers of neurons; an input layer, one or more hidden layers, and an output layer. This structure is shown in Figure 4.3.

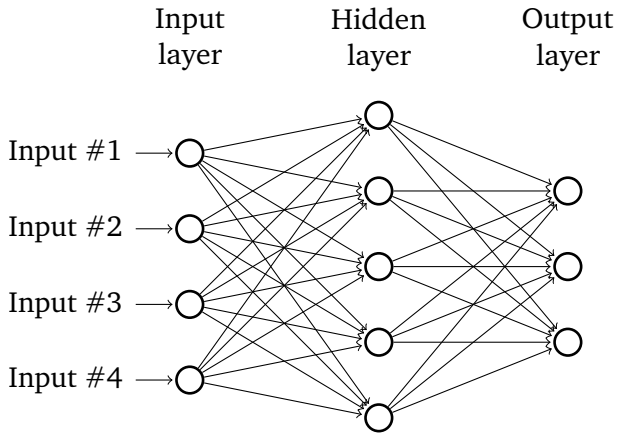


Figure 4.3: An example of a neural network with one hidden layer. The number of input neurons is 4, the number of hidden neurons is 5, and the number of output neurons is 3.

A trained neural network, works by feeding input data into the neurons in the input layer. The network then propagates this data through the network using the synapses, until the output neurons eventually contains a final result. This process of propagating input through the network, is known as forward propagation.

The basic idea of forward propagation is the following: Each input neuron has a given level of energy, which simultaneously spreads through the synapses to the hidden layer. Each synapse has a weight, determining the strengthening of the signal from an input neuron to a neuron in the hidden layer.

The input to a neuron n in the hidden layer, is the sum of the energy transmitted from neurons in the input layer, multiplied by the weights of the synapses from the input layer to n. This weighted sum, called the net input signal to n is then transformed by a so-called *activation function*, which yields the output signal from n. This principle of propagating energy continues, from the hidden layer to the output layer - or to the next layer of hidden neurons. To describe the process more formally, we introduce the

following notation[1]:

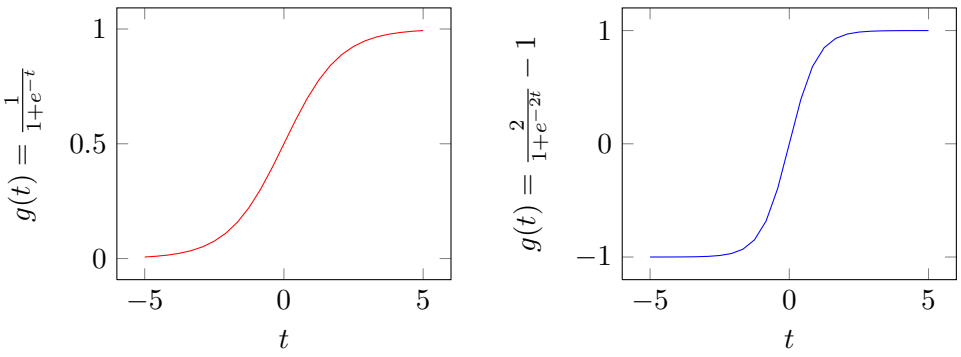
The j-th neuron computes a weighted sum a_j of its input as:

$$a_j = \sum_i w_{ji} z_i,$$

where z_i is the activation/output of neuron i and w_{ji} is the weight of the synapse connecting unit i to unit j. This weighted sum is then sent through an activation function g:

$$z_j = g(a_j),$$

to give the activation/output z_j of neuron j. The idea of the activation function is to squash/scale the net input energy, such that the output will be within a set of boundaries, and more importantly, the activation function introduces non-linearity into the model. A common choice of the activation function is the sigmoid function $g(t) = \frac{1}{1+e^{-t}}$. However, other possibilities include the hyperbolic tangent sigmoid activation function, $g(t) = \frac{2}{1+e^{-2t}} - 1$. Figure 4.4 shows the two activation functions.



Sigmoid activation function Tangent sigmoid activation function

Figure 4.4: Two of the most common activation functions for neural networks.

We write the total error E of the network as:

$$E = \sum_n E^{(n)},$$

where $E^{(n)}$ is the error on the n-th data sample. To minimize this error, we want to derive its gradient with respect to the weights. We apply the chain rule to obtain:

$$\frac{\partial E^{(n)}}{\partial w_{ji}} = \frac{\partial E^{(n)}}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}.$$

[1]We will ignore bias units in these derivations since they can be modelled by having an extra neuron in each layer with constant activation

We now introduce $\delta_j \equiv \frac{\partial E^{(n)}}{\partial a_j}$ called the errors. From $a_j = \sum_i w_{ij} z_i$ we get that

$$\frac{\partial a_j}{\partial w_{ji}} = z_i.$$

By using the definition of δ_j and the fact that $a_j = \sum_i w_{ij} z_i$, we can plug this into $\frac{\partial a_j}{\partial w_{ji}} = z_i$ to obtain that

$$\frac{\partial E^{(n)}}{\partial w_{ji}} = \delta_j z_i.$$

We can now write the errors for the output units as

$$\delta_k \equiv \frac{\partial E^{(n)}}{\partial a_k} = g'(a_k) \frac{\partial E^{(n)}}{y_k},$$

where we have used that $z_j = g(a_j)$ with z_k denoted by y_k.

To find the errors for the hidden units we apply the chain rule:

$$\delta_j \equiv \frac{\partial E^{(n)}}{\partial a_j} = \sum_k \frac{\partial E^{(n)}}{\partial a_k} \frac{\partial a_k}{\partial a_j},$$

where the sum is over all units k to which unit j is sending output. By plugging $\delta_j \equiv \frac{\partial E^{(n)}}{\partial a_j}$ into the above expression and using that $a_j = \sum_i w_{ij} z_i$ and $z_j = g(a_j)$ we get that

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k.$$

Finally we can write the gradients with respect to each training sample as

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^{(n)}}{\partial w_{ji}}.$$

Having determined the partial derivatives with respect to all weights, one may choose a nonlinear optimization algorithm – for example gradient descent – to adjust each weight.

4.4.1 Convolutional neural networks

Convolutional Neural Networks are a special type of neural networks, which have shown to be very successful in image classification tasks. The structure of the network is made such that it corresponds to overlapping parts of the visual cortex, thus modelling biological processes [Matsugu et al., 2003].

Commonly, convolutional neural networks have a structure alternating between so-called *convolutional layers* and *pooling layers*. The convolutional layers are tiled such

that they overlap, giving convolutional neural networks a tolerance for translation of the input image. The pooling layers simplify and combine the outputs of neighboring units, consequently reducing the resolution of the image and combining features. It is also possible to have fully connected layers in the network to model more complex interactions [Ciresan et al., 2012].

Recently, convolutional neural networks have achieved extremely high performance on image recognition tasks. They have achieved performance double that of humans on the problem of recognizing traffic signs and a misclassification rate of 0.23% on the MNIST dataset (digit recognition), which is the lowest ever achieved on the database to date [Ciresan et al., 2012].

Structure of convolutional neural networks

There is a wide variety of ways to construct a convolutional neural network. The most common structure is to alternate between convolutional layers and pooling layers, and then have a number of fully connected layers in the end of the network. One of the most well-known structures for convolutional neural networks is LeNet-5 developed in [Lecun et al., 1998], which is a network with 2 convolutional layers, 2 pooling layers and two fully-connected layers. The structure for LeNet-5 is shown in Figure 4.5 (which is taken from `http://deeplearning.net/tutorial/lenet.html`).

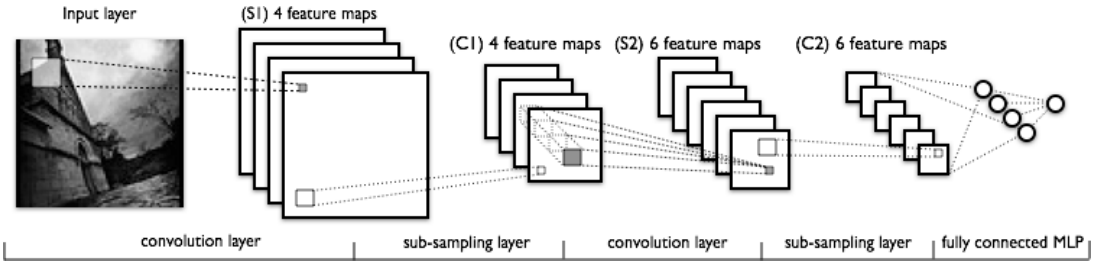
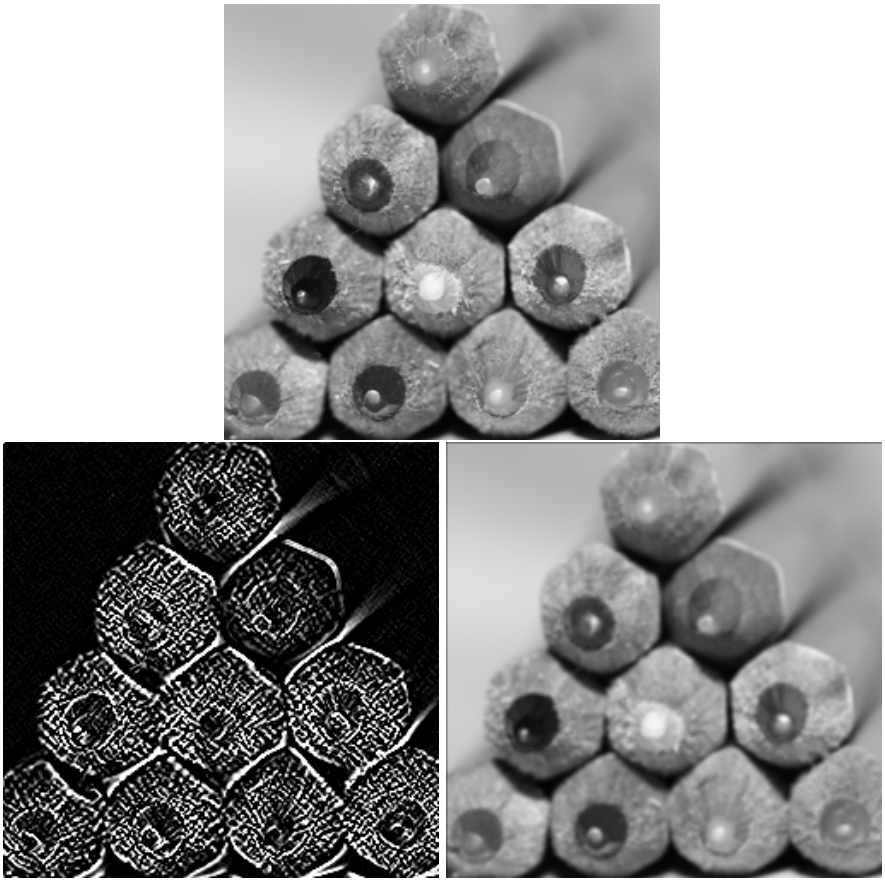


Figure 4.5: The network architecture for LeNet-5, consisting of 2 convolutional layers, 2 pooling layers (sub-sampling layers) and 2 fully connected layers. This depiction of the network is from `http://deeplearning.net/tutorial/lenet.html`

To understand how LeNet-5 works, we will describe conceptually how the different components work.

Convolution layers

A convolution layer works by having a set of so-called *kernels* which we convolve with the input image. A set of common kernels and their effect can be seen in Figure 4.6.



$$\begin{bmatrix} 10 & 0 & -10 \\ 0 & 0 & 0 \\ -10 & 0 & 10 \end{bmatrix} \qquad \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Figure 4.6: An image before convolution with a kernel, and the image after convolution with two different kernels. The first kernel works as a simple edge-detector and the second kernel blurs the image.

To convolve an image I with a kernel K, we do the following: For each pixel p in I, we place K on top, with its center on p. For each overlapping pair of pixels between the image and the kernel, we multiply their values together. Finally we sum all these products together, and the resulting sum is the new value of p (which is saved in a new image to not interfere with the other convolutions to be made on I). As an example, we show how to convolve a 3×3 kernel with a pixel in a 4×4 image:

$$\begin{bmatrix} 2 & 3 & 4 & 3 \\ 5 & \mathbf{2} & 1 & 1 \\ 1 & 4 & 5 & 4 \\ 2 & 2 & 3 & 3 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 3 & 4 & 3 \\ 5 & \mathbf{17} & 1 & 1 \\ 1 & 4 & 5 & 4 \\ 2 & 2 & 3 & 3 \end{bmatrix}$$

since

$$2 \cdot 0 + 3 \cdot 1 + 4 \cdot 0 + 5 \cdot 1 + 2 \cdot 2 + 1 \cdot 1 + 1 \cdot 0 + 4 \cdot 1 + 5 \cdot 0 = 17$$

A convolutional layer works by convolving the input image(s) with a number of kernels, adding a bias term and then applying a non-linear activation function (much like for normal neural networks) to obtain a number of new images (as many as the number of kernels).

When the input consists of multiple images, the applied kernels are 3-dimensional. This is illustrated in Figure 4.7 which is taken from the tutorial at `http://deeplearning.net/tutorial/lenet.html`. In the image, a single 3-dimensional kernel (with a total of 16 weights) is visualized, mapping 4 images to 2 images by doing convolution with 2 different 3-dimensional kernels.

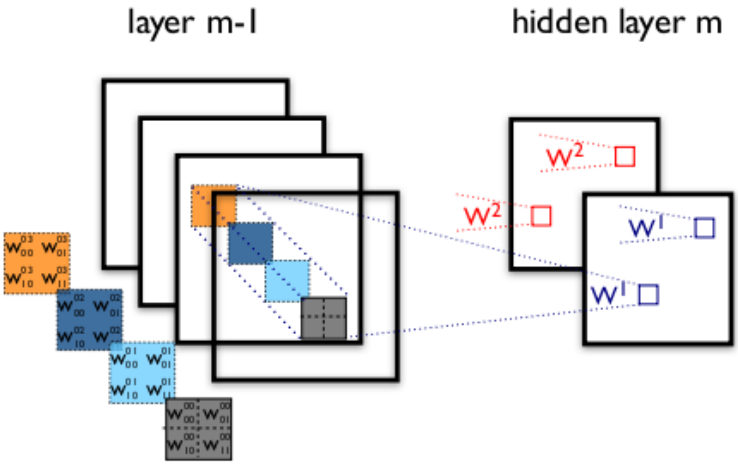


Figure 4.7: Example of a convolutional layer. This figure is from `http://deeplearning.net/tutorial/lenet.html`. Each kernel has 16 learnable parameters.

Pooling layers

The other new type of layer in a convolutional neural network, is the pooling (or sub-sampling layers). There are multiple types of pooling layers, where the two most common are average-pooling and max-pooling. Max-pooling which is non-linear partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value. Average-pooling (which is linear) works in the same way, but instead of outputting the maximum value, it outputs the average value. A pooling layer has a scaling factor, which determines how large the rectangles in the partitioning are, for example, a scaling factor of 2 will make the resulting image

4 times smaller as shown here for max-pooling:

$$\left[\begin{array}{cc|cc} 1 & 5 & 6 & 8 \\ 4 & 9 & 6 & 4 \\ \hline 4 & 8 & 2 & 1 \\ 1 & 9 & 5 & 4 \end{array}\right] \xrightarrow{\text{Max pooling}} \begin{bmatrix} 9 & 8 \\ 9 & 5 \end{bmatrix}$$

Chapter 5

Personalize Expedia Hotel Searches

This challenge posed by the online travel agency *Expedia*, asks researchers for help ranking hotels according to their likelihood of being booked. The description given for the competition was the following:

> “Expedia is the world’s largest online travel agency (OTA) and powers search results for millions of travel shoppers every day. In this competitive market, matching users to hotel inventory is very important since users easily jump from website to website. As such, having the best ranking of hotels (“sort”) for specific users with the best integration of price competitiveness gives an OTA the best chance of winning the sale.
>
> For this contest, Expedia has provided a dataset that includes shopping and purchase data as well as information on price competitiveness. The data are organized around a set of “search result impressions”, or the ordered list of hotels that the user sees after they search for a hotel on the Expedia website. In addition to impressions from the existing algorithm, the data contain impressions where the hotels were randomly sorted, to avoid the position bias of the existing algorithm. The user response is provided as a click on a hotel or/and a purchase of a hotel room.
>
> Appended to impressions are the following:
>
> 1. Hotel characteristics
> 2. Location attractiveness of hotels
> 3. User’s aggregate purchase history
> 4. Competitive OTA information
>
> Models will be scored via performance on a hold-out set.

5.1 The dataset

The dataset is split into two parts, a training set and a test set. The training set consists of 336,334 search queries, each containing between 4 and 38 results, and totalling to 9,917,530 search results combined. The test set consists of 266,230 search queries, containing between 5 and 37 results and totalling to 6,622,629 search results.
As shown in Figure 5.1, a user inputs a number of criteria on Expedia's website to perform a query for a hotel. The website then returns a ranked list of hotels - ranked by Expedia's own ranking algorithm. A tiny fraction of the searches are returned in a random order to help train their ranking algorithm. A hotel can then be clicked if the user wants more information, and booked if the user is satisfied. For some queries, the price from competing travel agencies is known.

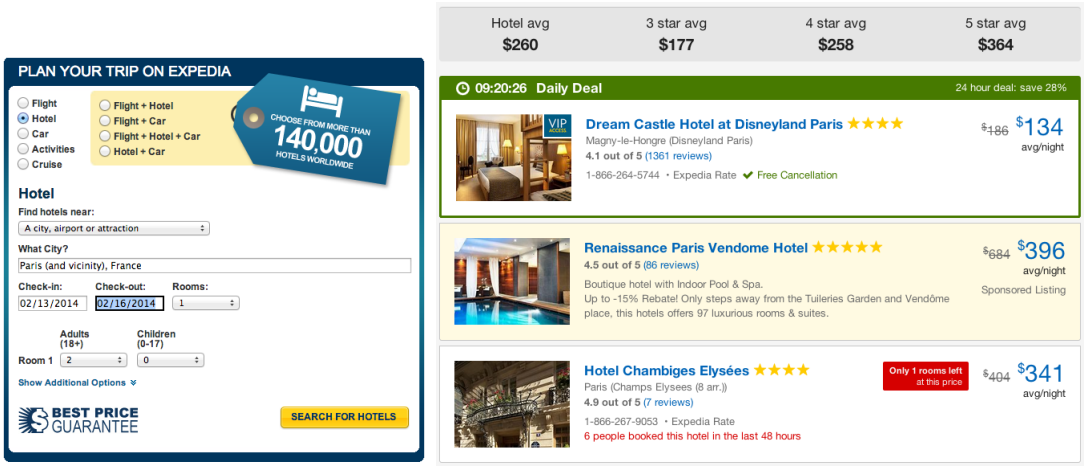


Figure 5.1: In the left subfigure a search query for a hotel in Paris is shown. The user indicates the location of the stay, the dates, the number of rooms and the number of people. In the right subfigure the three top results for the query is shown. The price per night is indicated together with the star rating and the review score and additional information about the hotel.

The original training data includes 54 variables of generally 5 different types:

- Criteria in the search query
- Static hotel characteristics
- Dynamic hotel characteristics
- Visitor information
- Competitive information

We will give a short highlight of the different types of features, and look more in depth at the more interesting ones:

Criteria in the search query These variables include: *Date and time of search, the destination id, length of the stay, booking window (how much in advance the booking was made), number of adults, number of children, number of rooms, whether the stay was short and if it included a Saturday night*.

Static hotel characteristics These variables include: *Hotel id, hotel country id, star rating, user review score, whether the hotel is independent or part of a chain, desirability of hotel location (two variables for this), historical information about the hotel price*.

Dynamic hotel characteristics These variables include: *Rank position (how far down the list the hotel was shown), price, whether there was a promotion, whether the hotel was clicked, whether the hotel was booked, total value of transaction, probability that the hotel will be clicked on in an internet search (obtained from advertisement data, like Google adwords)*.

Visitor information These variables include: *Country id for visitor, distance between visitor and hotel, mean star rating for visitor, mean price per night for visitor*.

Competitive information These variables include: *If Expedia has lower/same/higher price than competitor, if the competitor has hotel availability, price difference*. These variables were given for 8 specific but unknown competitors. The information about competitors was automatically gathered by Expedia when possible.

Other Besides the above mentioned variables, there was a *search id, a variable indicating whether the search results were returned in random order and an id for the site (expedia.com, expedia.dk, etc.) which the query was made on*.

5.1.1 The leaderboard data and test variables

The 266,230 search queries, making up the 6,622,629 samples in the test set is split into a public (which is shown on the leaderboard) and a private test set (for the final evaluation). The split is 25% for the public part and 75% for the private part, where the split is made by randomly selecting 25% of the search queries by id.

In the test data, the following information is hidden: *Whether a hotel was booked, whether a hotel was clicked, the total value of the transaction and at which position the hotel was shown on Expedia's page*.

5.2 Evaluation metric

For evaluation in this competition, the metric was the average NDCG@38 (normalized discounted cumulative gain) over all queries (see Section 4.2.3 for a description of the NDCG@k error metric). For this competition, the relevance of a hotel which was

booked was set to 5, the relevance of a hotel which was clicked on but not booked was set to 1, and a relevance of 0 was chosen for the hotels which were not clicked on. k was chosen to be 38, the maximum number of search results in any query in the dataset. This was a decision made by the competition organizers.

As an example, say we have 5 hotels $H_1, \ldots, H_5$ which we decide to rank in the order H_2, H_5, H_1, H_3, H_4. Say also that in reality, the hotel H_1 was booked, and that the user clicked on H_2 but did not book it. Consequently we have $\text{rel}_3 = 5$ and $\text{rel}_1 = 1$ since the hotel we ranked third had a relevance score of 5 (it was booked) and the hotel we ranked first had a relevance score of 1 (it was clicked). Now we can compute the DCG@38 for the ordering as

$$\begin{aligned}\text{DCG}_{38} &= \frac{2^1-1}{\log_2(2)} + \frac{2^0-1}{\log_2(3)} + \frac{2^5-1}{\log_2(4)} + \frac{2^0-1}{\log_2(5)} + \frac{2^0-1}{\log_2(6)} + \left[\frac{2^0-1}{\log_2(7)} + \ldots + \frac{2^0-1}{\log_2(39)}\right] \\ &= \frac{2^1-1}{\log_2(2)} + \frac{2^5-1}{\log_2(4)} \\ &= \frac{1}{1} + \frac{31}{2} = 16.5,\end{aligned}$$

where all terms with $i > 5$ has relevance 0 and consequently adds nothing to the DCG. To compute the IDCG, we order the hotels in an optimal fashion, for example H_1, H_2, H_3, H_4, H_5 which gives

$$\text{IDCG}_{38} = \frac{2^5-1}{\log_2(2)} + \frac{2^1-1}{\log_2(3)} + \left[\frac{2^0-1}{\log_2(4)} + \ldots + \frac{2^0-1}{\log_2(39)}\right] = \frac{31}{1} + \frac{1}{\log_2(3)} \approx 32.58.$$

This gives an NDCG@38 score of

$$\text{NDCG}_{38} = \frac{\text{DCG}_{38}}{\text{IDCG}_{38}} \approx \frac{16.5}{32.58} \approx 0.506.$$

5.2.1 Challenges with the data and interesting observations

There were a number of challenges when working with the dataset. In this section we will mention some of these challenges along with some interesting observations.

Size of the dataset

The first issue with the dataset is the size. Simply loading the training data into memory with R requires almost 10GB of memory. Thus working with the entire dataset is a challenge when training complex models.

Multiple output variables

An important issue to be adressed with the data is the special target values and the corresponding evaluation criteria. Hotels which have been booked gets a relevance score of 5, and hotels which have only been clicked gets a relevance score of 1. If training a model which is not directly optimizing the correct evaluation criteria (NDCG@38), one needs a strategy for handling these two different outputs with different weights.

Imbalanced data

In each search query, at most one hotel has been booked, and often only very few have been clicked. This means that most samples in the dataset are negative (hotels not preferred by the user). When training statistical models with this data, one has to be careful that the models do not simply learn to always predict that the hotel was not booked.

Another common issue with an imbalanced dataset is that if one trains a classifier outputting probabilities, these might be very skewed (small) for the positive class (if that class is under represented). Note that in this competition this is not a problem since we are dealing with a ranking problem, where we are only interested in knowing the most probable hotel booking, not how probable it is.

Categorical variables

While most of the features are numerical, a number of them are categorical id-features. The categorical features are: *Id for the search query, id for the hotel property, id for the site (expedia.com, expedia.dk etc.), id for the country of the visitor and an id for the search destination (usually at city level)*. None of these categorical variables have any explanatory labels attached to them, and the id-numbers reveal no information about their meaning.

Since the id of the search query does not hold any information (assuming that there is no data leakage), there is no reason to expand this variable. There are thousands of properties, so expanding their id's is not feasible. The three last categorical variables *site id, visitor country id and property country id* has between 30 and 130 different values – so expansion is possible but will grow the size of the dataset by a factor of 5-6 (not taking into account the sparsity of these features).

Temporal information

Since the data consists of bookings made at specific times throughout the year, it is interesting to see if one can use the temporal information in the data. In Figure 5.2 and Figure 5.3 we have plotted how the data is distributed in time over the year. Some clear trends to see is that a lot of travelling happens around Christmas and New Year's Eve, and that people travel more frequently during the weekends.

Position bias

The results returned from a user query is presented in a list. In most cases (around 70%) this list is sorted by Expedia's own ranking algorithm. In the rest of the cases the results are shown in a random order (this is done purely for providing information to Expedia about the effectiveness of their ranking and about the position bias). In the dataset, it is stated whether a query was presented to the user in random order, and in which order the hotels were shown. In the test set, the ordering is not given, but

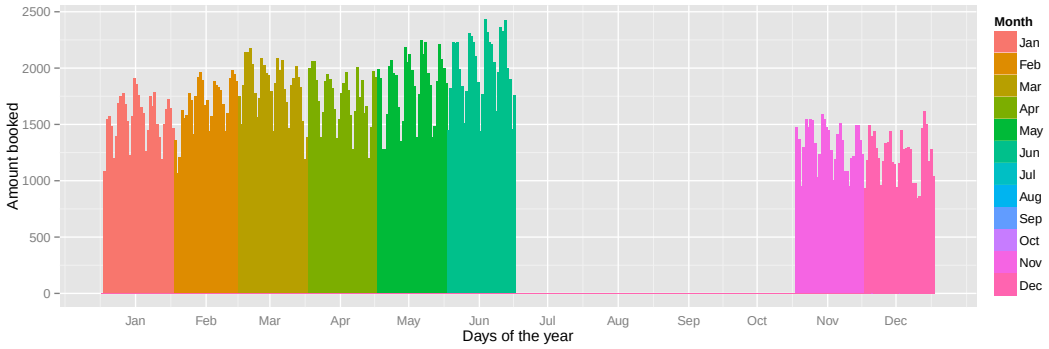


Figure 5.2: This plot shows for each day in the training set, how many hotel bookings were made on that specific day (the day the user made the actual booking online).

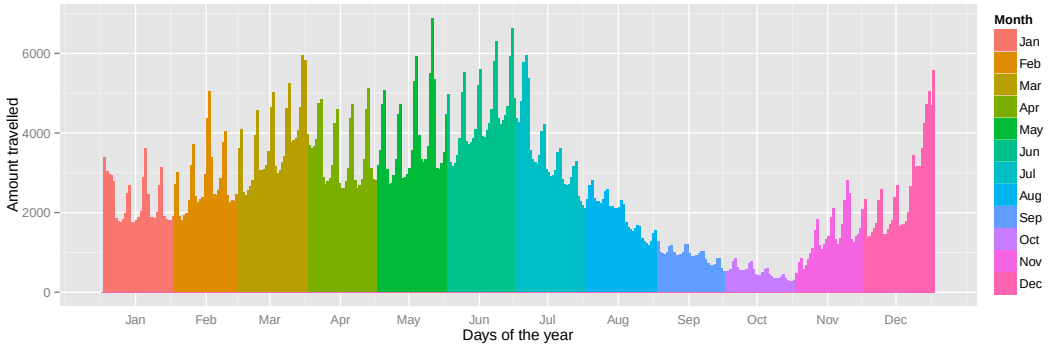


Figure 5.3: This plot shows for each day in the training set, how many hotel bookings include that specific day. Note that this includes the months July, August, September and October because even though no bookings were made in these months, some of the earlier bookings were for this period.

whether the hotels were shown in random order or ranked by Expedia's algorithm is known.

Since Expedia's algorithm is performing better than random, there is a clear correspondence between the position of a hotel on the list of results, and the probability that the hotel was clicked on or booked. What is not as clear, is the amount of bias introduced by the positioning, even if the hotels are presented in random order. To give an idea of the amount of so-called *position-bias*, Figure 5.4 shows the likelihood of hotels being clicked on or booked given their position in the resulting list.

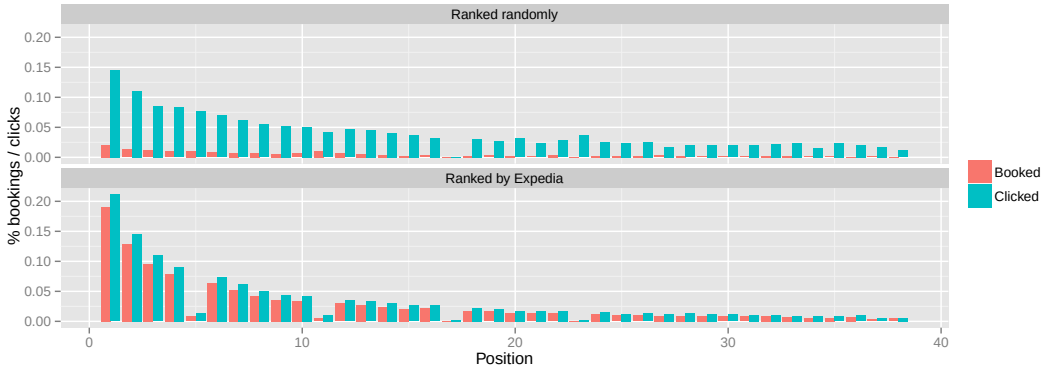


Figure 5.4: This plot shows the distribution of bookings and clicks over the different positions on the returned ranking. In the top part of the plot the results were permuted randomly and in the bottom part of the plot the results were ranked by Expedia's own algorithm. From this plot we can see that there is still a bias towards picking the results in the top even if the results are randomly permuted.

Outliers

Since the data from Expedia is gathered from their databases without manually looking through it, the threat of having outliers in the data should be considered. In Figure 5.5 a box-plot is shown for a subset of the variables indicating that some of the variables in fact do have outliers.

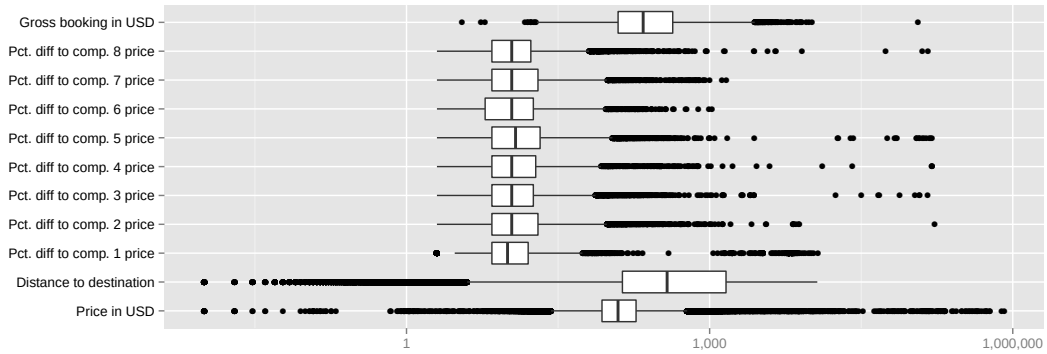


Figure 5.5: This plot shows the distribution of values in certain features. Looking at the way the values are distributed, many of them appear to be outliers.

Missing data

In this challenge, one of the important hassles is the amount of missing data. In Figure 5.6 a bar plot shows how big a portion of each variable is missing in the dataset. We observe that quite a few variables have around 90% missing data, which has to be handled when dealing with the data and should be taken into consideration when choosing and training statistical models.

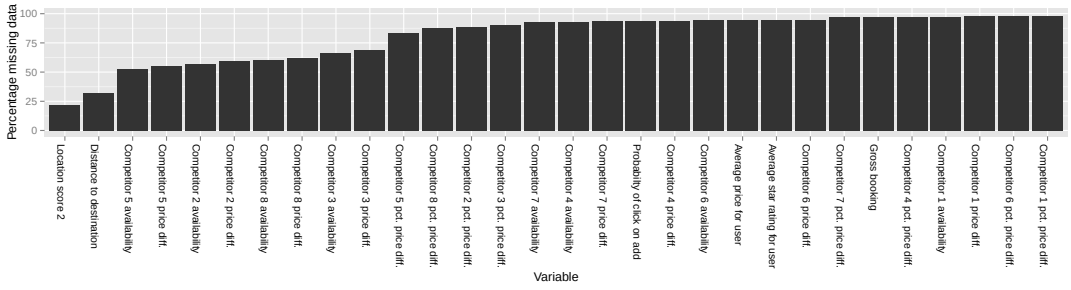


Figure 5.6: This plot shows the amount of missing data for some of the variables in the dataset.

Competitor variables

An initial hypothesis might be that competitors prices are not of importance when deciding on whether to book a hotel online. To get a quick intuition about the validity of this claim, one can look at Figure 5.7 showing that the prices of competitors do in fact tell us something about the probability of a hotel being booked or clicked on. It is possible though, that this is due to confounding variables either hidden or in the dataset.

5.3 Approach and progress

In this subsection, we will try to outline and explain the modelling process for the Expedia problem. The content in this subsection is written chronologically to give an understanding of the decisions and insights we got while trying to solve this problem. When mentioning a submission to the leaderboard which achieves a best score so far, we will mark this score with bold.

5.3.1 Setting up the pipeline

We started by downloading the data, reading through the competition description and by making a well-structured folder structure for all files generated in this competition. We then went on to look at the training data, plotting various things to get a better understanding of the different features (some of these plots are the ones that we have touched up for Figures 5.3 to 5.7).

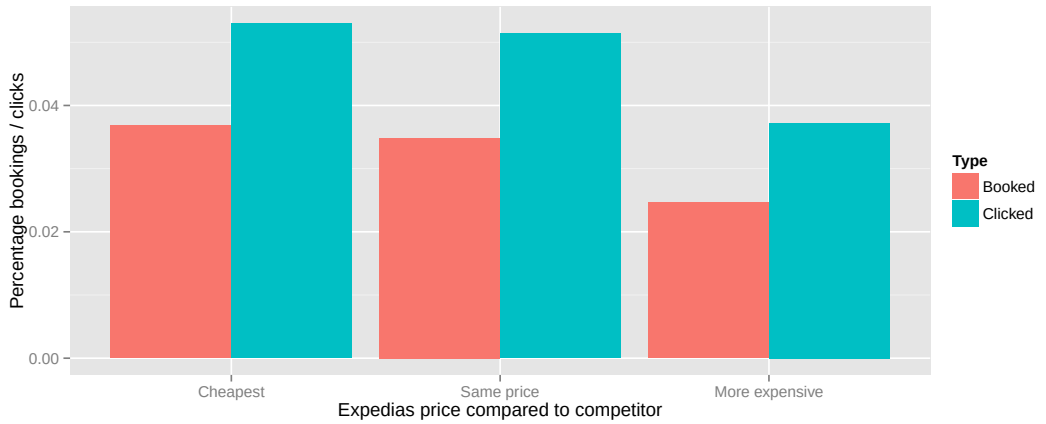


Figure 5.7: This plot shows the percentage of bookings as a function of Expedia's price compared to the competitors prices.

5.3.2 Python benchmark

To get started on the competition, we decided to run and submit a benchmark code released by the competition administrators. The benchmark written in Python reads in the first 100,000 samples from the training data and trains a random forest with 50 trees. The random forest is trained to predict the value of the variable "booking_bool", indicating if a hotel was booked. The model was used to predict a probability of being booked for each hotel in the test set, and the queries were sorted by this probability. This benchmark model got a NDCG-score of **0.40356** on the leaderboard.

5.3.3 Price benchmark

Some even simpler benchmarks were attempted to get a feeling for the predictiveness of the different variables. One idea was to simply sort the queries by price, putting the cheapest hotel in the top – this giving a leaderboard score of **0.37893**. Another idea was to sort by the variable "location_score_2" which had the highest correlation with whether a hotel was booked – this giving a leaderboard score of **0.37922**.

5.3.4 RankLib

Since the problem is a learning to rank problem, we looked to RANKLIB [Dang, 2011] (now a part of the Lemur project) which is a library of learning to rank algorithms, including LambdaMART (described in Section 4.2), a listwise algorithm which can directly optimize the NDCG-metric used for this competition. RANKLIB is written in Java and uses a special file format for the input.

We wrote a MATLAB-script for loading in the data, preprocessing it and then writing it in a format readable by RANKLIB. We then trained a LambdaMART ensemble with

1000 trees and we stopped training after 10 iterations in a row with no improvement on a validation set consisting of 10% of the training data. This lead to a leaderboard score of **0.44528**.

5.3.5 Conversion of prices

For some countries in the data, the price stated for at booking is per day, and for some countries the price is for the entire stay. To get rid of this discrepancy, we converted all prices into per-day prices. It was not specified which countries had per day prices and which had per stay prices, so we inferred this ourselves. To determine if a specific country had per stay or per day prices, we looked at the booking price as a function of the booking duration, and determined if there was a clear growth. Some of these trends were spotted manually since the data was quite noisy. All prices which were given as per stay were converted to per day by dividing the price by the booking duration.

5.3.6 Construction of new features

We tried to create a new set of features which would improve the accuracy of our model. In total, more than 30 new features were added. We will describe which features were made and for some of them give a short intuition for why this feature might be relevant.

We started by splitting the date-time variable into six variables: second, minute, hour, day, month and year. The date-time variable indicated when the query was made and was formatted as a string. The rationale behind keeping some time information was that specific types of queries might be made at specific times of the day or in specific parts of the year.[1]

We made a boolean variable indicating whether the user performing the query had any purchase history or not. Returning users might have a different way of shopping than new customers.

We made some new variables which were averages, medians and standard deviations over the property-id's. The variables we computed statistics for were: The star rating, the review score, the location score (both of them) and the price. The rationale behind computing these summary statistics over a property was mostly to lower the noise in the data.

We made a variable indicating whether the user making the query was located in the country which was most common compared to the site (.jp, .com, etc.) or not. For example, people in USA would have a 1 if they used .com, whereas people from Japan would have a 1 if they were using .jp. The idea behind this was that people booking hotels while abroad might have different shopping patterns. The most common site id

[1]The time of the query in the data was the local time at the computer making the query.

for each country id was inferred from the data.

We made a variable indicating the difference between the hotels price and the average price in the country it is in. Since the hotels are only compared with other hotels within the same query, this variable should not have much importance, but might be a bit better because of its normalization.

We made two variables combining other variables about the hotel and query. One for the total number of people in the booking and the number of people per room. We also made variables for the price per person and the price per adult. Since some models are not able to directly combine variables, such combination variables might help.

For some users we have historical information available, in the form of an average purchase price and an average number of stars for the hotels previously booked. Using this information, we created two new variables with the difference between a hotel's price and the user's average purchase price, and the difference between the hotel's star rating and the average number of stars for the hotels in the users purchase history.

Two of the variables were given after having gone through a transformation. The first one is "prop_log_historical_price" which is the logarithm of the mean price of the hotel over the last trading period. The other is "srch_query_affinity_score" which is the log of the probability a hotel will be clicked on in Internet searches. We created two new variables by exponentiating "prop_log_historical_price" and by taking 10 to the power of "srch_query_affinity_score"[2].

Some of the hotels in the dataset have been shown for multiple queries. To take advantage of historical information about the popularity of a hotel, we create a new variable, indicating how often a hostel is clicked as a percentage of all its impressions.

Finally we created 5 variables which compared hotels within a query to each other. For each hotel, we created new variables indicating how it related to the other hotels in the same query. We did this by calculating an absolute difference and a relative difference (in percentage) between the hotel and the average hotel (on a set of numerical variables) within the same query. The variables used for this were the price, star rating, review score, location score 1 and location score 2. This gave in total 10 new variables.

5.3.7 Removal of outliers

As described in Section 5.2.1, the data given had a number of outliers for specific features. The most extreme such case was a number of extremely high hotel booking prices in the dataset. By manually looking at the data, we decided to remove all training samples where the price was more than 10,000 USD.

[2]The bases for the exponents were revealed on the Kaggle forum in the thread at `http://www.kaggle.com/c/expedia-personalized-sort/forums/t/5784/log-base-for-srch-query-affinity-score`

5.3.8 Inference of missing data

Another big issue mentioned in Section 5.2.1 is the amount of missing data in the dataset. For a number of variables, the amount of missing data is above 90% and around half of the features have some amount of missing data.

We attempted to infer the missing data in the training set before model training and in the test set before making predictions. The way we inferred the missing data differs for the different variables types. We had separate approaches for the variables "orig_destination_distance", "prop_location_score_2" and "prop_review_score" and a general method for the rest.

For the feature "orig_destination_distance" containing the distance between the user and the hotel location, we built a matrix containing pairwise distances between the countries in the dataset. Using this distance matrix we could then make approximative guesses for the distance between the user and the hotel. With this approach, we were able to infer a little less than half of the missing distances (1,501,849 of the original 3,216,461 missing values were inferred). The rest of the missing values were imputed with the median.

For the feature "prop_location_score_2" we had around 20% missing data, but no data was missing for the feature "prop_location_score_1", and these two variables had a high correlation (0.53 spearman correlation). The location scores also had some correlation with the price for the hotel (0.16 spearman correlation between location score 2 and the price). Using "prop_location_score_1" and the price for the hotel, we built a linear regression model to predict "prop_location_score_2".

For the feature "prop_review_score" we again built a linear model, this time using the variables "prop_starrating", "price_usd", "prop_location_score1", "prop_location_score2" and "prop_brand_bool" which all had some predictive power for the review score. Using this linear model, we inferred estimates for the review score.

For the rest of the missing data, we simply imputed the median value for the feature. The median was chosen instead of the mean to avoid outliers having too large an effect on the imputed value.

5.3.9 Feature removal

Before training the statistical models, we removed the features "minute" and "second" because there was no intuitive reason to keep them in the dataset. The reason for removing them was to reduce the amount of irrelevant features and consequently the amount of noise in the data. Besides these features, we removed the features which were not included in the test set.

5.3.10 Submissions

At this point we again tried training LambdaMART models with the new features, but without good results. Using validation set of 10% we obtained a NDCG of 0.4628 on the local validation set, but this only resulted in 0.38596 on the leaderboard. After unsuccesfully trying to find the error and two more submissions to the leaderboard with bad results, we decided to rewrite the data-processing code to eliminate mistakes.

5.3.11 Rewriting of data munging code to R

To make sure the data-processing was done correctly, we decided to port the code from MATLAB to R. The decision of using R was made because it has some advantages when working with data which is not in a simple matrix-format (where MATLAB is advantageous).

Converting the dataset to the format used by RANKLIB was extremely time consuming, and the same was the case for the training time for the LambdaMART models. In total it took approximately 2-3 days to construct the data, convert it to the right format and train a model. At this point there was only around 14 days left of the competition, so we decided to try simpler models which would enable us to iterate faster.

5.3.12 Modelling with a random forest

Using the implementation of Random Forests in the R-package `randomForest`, we trained a Random Forest with 100 trees on 10% of the training data. The trees in the random forest were regression trees, where the target variable was 0 if a hotel was neither booked nor clicked, 1 if the hotel was clicked on and 5 if the hotel was booked. This gave a NDCG-score of **0.46733** on the leaderboard, taking us up more than 30 positions. Training once more with 500 trees improved the score slightly to **0.46809**. The rationale behind using random forests as a predictor was two-fold: On one hand, random forests have previously shown exceptional results in Kaggle competitions[3], and additionally, the decision trees – which random forests are built of – resemble human decision making which we were trying to model. By further removing a set of features which were highly correlated, the leaderboard score was increased to **0.47285**.

5.3.13 Clustering of data

Up until this point, we had not used the categorical variables *id for the site (expedia.com, expedia.dk etc.), id for the country of the visitor and an id for the search destination)*. The reason was that they had no meaningful labels (they were just random integers), and that expanding them into a large set of binary variables would make the dataset too large to easily handle.

[3]As described on the Kaggle wiki (19th of February, 14:58): `http://www.kaggle.com/wiki/RandomForests`

Intuitively, the origin of the customer could be an important factor. One could imagine situations where a cultural difference between countries might lead to different shopping patterns. To utilize this information in the data, we tried building multiple models. Instead of building a separate model for each of the 202 countries in the dataset, we created a number of clusters, and built a model for each cluster. To construct the country-clusters we used the variables "orig_destination_distance", "price_relative_to_location", "usd_pr_night_pr_adult", "prop_brand_bool", "total_people", "srch_children_count", "prop_location_score1", "prop_location_score2", "prop_review_score", "prop_starrating", "srch_booking_window" and "srch_saturday_night_bool". Using these features, we used k-nearest neighbor clustering with 10 clusters to find the most meaningful clusters. Manually inspecting the way the country-ids were distributed between the clusters indicated that the clusters had captured some meaningful information differing between countries.

Using the found clusters, we trained a random forest for the samples in each cluster. When predicting on a new data sample x, we used the model corresponding to the cluster with the center closest to x. Unfortunately, the approach of training multiple models on clusters did not yield any results which improved the leaderboard score.

5.3.14 Logistic regression with regularization

We tried using a simple logistic regression model with L_2-regularization (as described in Section 4.3) and the same target values as for the random forest, giving a small increase in score to **0.47371** on the leaderboard. The final model was a so-called *elastic-net* which combines L_1 -and L_2-regularization, trained with 20% of the training data. This gave a final score of **0.47536** on the leaderboard.

5.4 Results

When the competition ended, our submission ranked 138th out of a total of 337 submissions. The final NDCG score on the private test set was 0.47301, where the winning submission got a score of 0.54075 and Expedias own algorithm obtained a score of 0.49748 (corresponding to a 65th position on the private leaderboard).

To evaluate why our approach was not more succesful than it was, one can look at the Top 3 submission, and see what they did. The following sub sections are summaries of their approaches, taken both from the forums at Kaggle and from a set of presentations found at `https://www.dropbox.com/sh/5kedakjizgrog0y/_LE_DFCA7J/ICDM_2013`

5.4.1 Michael Jahrer and Andreas Toescher, 1st place

Their single best model was a LambdaMART model from rankLib, achieving a score of 0.5338. As features, they used:

- All numeric features

- Average of numeric features per prop_id
- Standard deviation of numeric features per prop_id
- Median of numeric features per prop_id

The last three of those were calculated using the training and test set. The rest of the model blend was variations of rankLib/LambdaMART, Gradient Boosted Decision Trees (getting 0.5256 on leaderboard), Neural Networks (getting 0.5297 on leaderboard) and Stochastic Gradient Descent models (getting 0.50377 on leaderboard).

5.4.2 Owen, 2nd place

The second place submission got a score of 0.5398 on the private leaderboard.

Missing values were imputed with negative values. Numerical values were capped/bounded to remove outliers. Negative instances were down sampled giving faster training time and better predictions.

The features used were

- All original features
- Numerical features averaged over
  - srch_id
  - prop_id
  - destination_id
- Composite features, such as
  - Price difference from recent price
  - Price relative to others in query
- Estimated position
- EXP features (expected features)

EXP Features is a way of converting categorical features into numerical features. Each category in a categorical feature is replaced with an average of the target variable (booked/clicked) related, excluding the current observation. The estimated position was computed like an EXP feature using the property id, the destination id, the target month and the position of the same hotel in the same destination in the previous and next search.

The model used was an ensemble of Gradient Boosting Machines with NDCG as the loss function (implemented in R). Two types of models were trained, some with the

EXP features, and some without the EXP features.

The most important features were:

- Position
- Price
- Location desirability (ver. 2)

5.4.3 Jun Wang, 3rd place

The third place submission got a score of 0.53839 on the private leaderboard.

A hypothesis was that users do not like to book hotels with missing values, consequently missing values were filled with worst case scenario. The missing values of competitor descriptions were all set to zero.

Features were normalized with respect to different indicators (for example the search id, property id, month, booking window, destination id and property country id).

The models trained were LambdaMART, SVM-Rank and linear regression. The final model was a LambdaMART model with 300 features.

5.5 Discussion

A position of 138 out of 337 on the leaderboard was not as good a result as we hoped for in this competition.

5.5.1 Things we did right

Many of the features we created were also created by the top participants. Generally, feature engineering seemed to be an important part of this competition.

The attempt to use LambdaMART as a model seems like a good idea. Two of the Top-3 competitors used LambdaMART as their final model. The second place submission was made with Gradient Boosting Machines, optimizing NDCG. The take-away from this is – like stated in Section 2.5 – that predicting the right thing is important.

5.5.2 Things we did wrong

Allthough we did a lot of feature engineering, the winning submission made even more features which were averages over different categorical variables. These variables seemed to have an important effect on making their models so good.

While fairly complex models were the ones used for the winning submissions, slightly simpler models were also able to achieve high scores – with much shorter training time.

By looking at the write-ups from the winning competitors, their approach to imputing missing values were different from ours. They imputed with worst-case values, which might have made a big difference.

Our models did not take the position bias into account, and this was most likely an important part of building a winning model.

At least one of the winning submission made use of undersampling. This gave faster training times and better models.

5.5.3 Lessons learned

The main lessons to take away from this competition are about missing values, undersampling, feature engineering and predicting the right thing. It should also be noted that the winning solutions made use of ensembling – supporting the hypothesis in Section 2.3.

Chapter 6

Galaxy Zoo - The Galaxy Challenge

This challenge posed by Galaxy Zoo and Winton Capital asks the participants for help classifying galaxies according to their shapes. The description given for the competition was the following:

> “Understanding how and why we are here is one of the fundamental questions for the human race. Part of the answer to this question lies in the origins of galaxies, such as our own Milky Way. Yet questions remain about how the Milky Way (or any of the other ~100 billion galaxies in our Universe) was formed and has evolved. Galaxies come in all shapes, sizes and colors: from beautiful spirals to huge ellipticals. Understanding the distribution, location and types of galaxies as a function of shape, size, and color are critical pieces for solving this puzzle.
>
> With each passing day telescopes around and above the Earth capture more and more images of distant galaxies. As better and bigger telescopes continue to collect these images, the datasets begin to explode in size. In order to better understand how the different shapes (or morphologies) of galaxies relate to the physics that create them, such images need to be sorted and classified. Kaggle has teamed up with Galaxy Zoo and Winton Capital to produce the Galaxy Challenge, where participants will help classify galaxies into categories.
>
> Galaxies in this set have already been classified once through the help of hundreds of thousands of volunteers, who collectively classified the shapes of these images by eye in a successful citizen science crowdsourcing project. However, this approach becomes less feasible as datasets grow to contain of hundreds of millions (or even billions) of galaxies. That’s where you come in.

> This competition asks you to analyze the JPG images of galaxies to find automated metrics that reproduce the probability distributions derived from human classifications. For each galaxy, determine the probability that it belongs in a particular class. Can you write an algorithm that behaves as well as the crowd does?

6.1 The dataset

The data consists of 141,553 images of galaxies, where 61,578 of them are in the training set. The leaderboard is 25% of the test set and the remaining 75% of the test set is used for the final evaluation. The images are 424×424 pixels and they are in colors. Examples of images from the training set are shown in Figure 6.1.

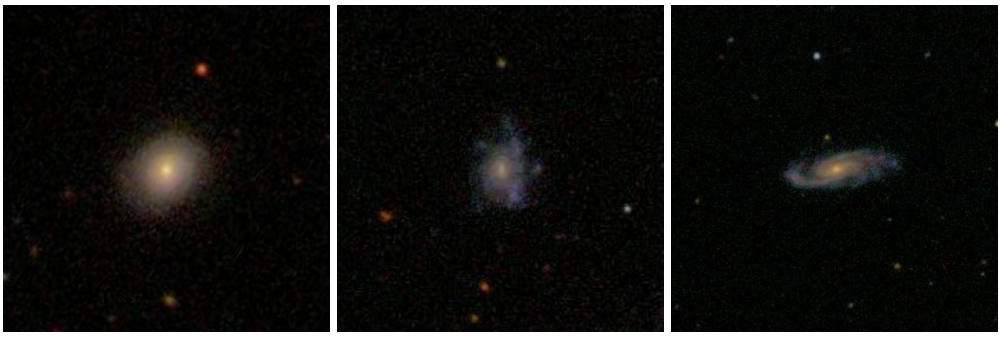


Figure 6.1: Three galaxies with ID's 100008, 100134 and 100380 from the training set. Each image in the dataset contains a galaxy or some artifact in the center of the image.

The task is to classify a galaxy into a set of categories. For each galaxy, a number of human users have been asked a series of questions used to classify the galaxy. Which questions are asked depends on the answers to previous questions, so the possible questions can be described by a decision tree. This decision tree consists of 11 questions with a total of 37 different answers. The decision tree is visualized in Figure 6.2 and the questions/answers are given in text form in Table 6.1.

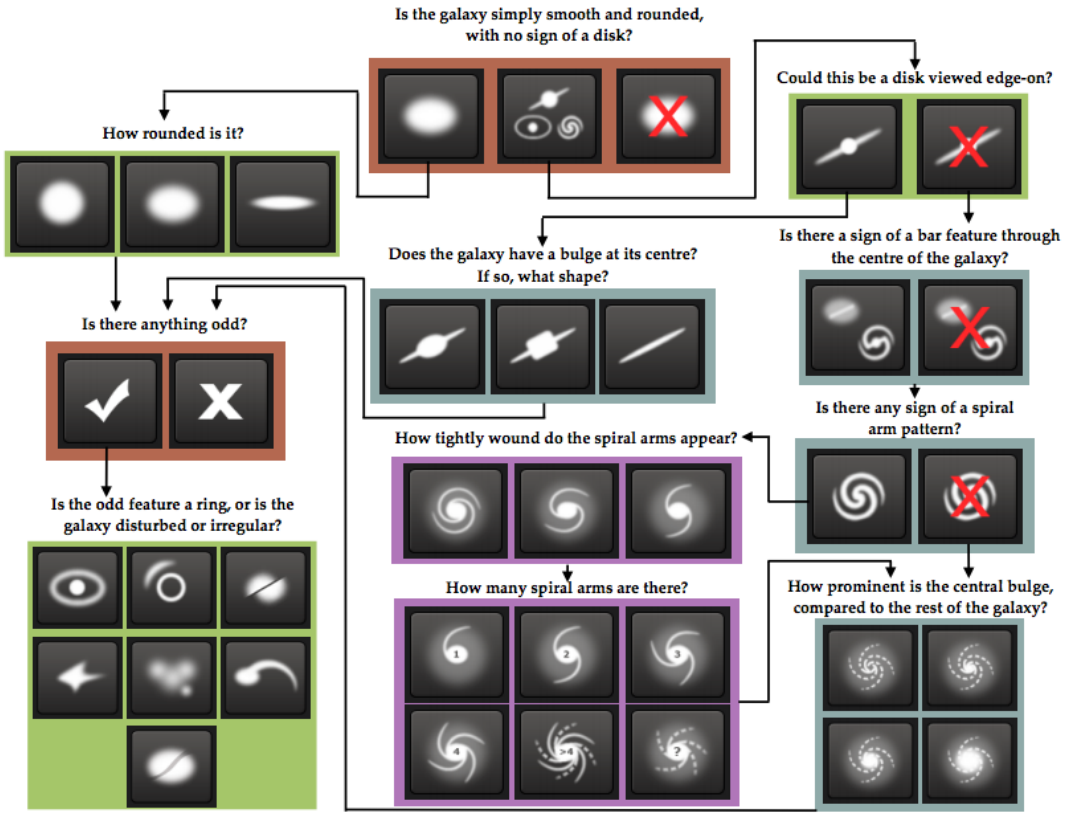


Figure 6.2: The decision tree the users are taken through when classifying a galaxy on GalaxyZoo. This figure is taken from [Willett et al., 2013].

The task of classifying galaxies is not easy, and often the human classifiers disagree on the questions in the decision tree. To quantify these disagreements, we are given data on how the users distribute on the possible answers. The task is to predict how the users of Galaxy Zoo distribute on the possible answers to each question. Say that for a specific galaxy, 20% answers that the galaxy is "smooth", 60% answers it is "features or disk" and the last 20% answers that it is "star or artifact", then the first three values in the output for that galaxy will be

$$[0.2,\ 0.6,\ 0.2].$$

To put priority on the most important questions in the decision tree which determines the overall structure of the galaxy (the questions in the top of the tree), the later questions are down-weighted. Continuing the example above, look at the 60% who answered "features or disk". These 60% now go to Question 2, and say that 70% answers "yes" and 30% answers "no". Then the next two outputs for this galaxy will be

$$[0.6 \times 0.7,\ 0.6 \times 0.3] = [0.42,\ 0.18],$$

Task	Question	Responses	Next
01	*Is the galaxy simply smooth and rounded, with no sign of a disk?*	smooth	07
		features or disk	02
		star or artifact	**end**
02	*Could this be a disk viewed edge-on?*	yes	09
		no	03
03	*Is there a sign of a bar feature through the centre of the galaxy?*	yes	04
		no	04
04	*Is there any sign of a spiral arm pattern?*	yes	10
		no	05
05	*How prominent is the central bulge, compared with the rest of the galaxy?*	no bulge	06
		just noticeable	06
		obvious	06
		dominant	06
06	*Is the anything odd?*	yes	08
		no	**end**
07	*How rounded is it?*	completely round	06
		in between	06
		cigar-shaped	06
08	*Is the odd feature a ring, or is the galaxy disturbed or irregular?*	ring	**end**
		lens or arc	**end**
		disturbed	**end**
		irregular	**end**
		other	**end**
		merger	**end**
		dust lane	**end**
09	*Does the galaxy have a bulge at its centre? If so, what shape?*	rounded	06
		boxy	06
		no bulge	06
10	*How tightly wound do the spiral arms appear?*	tight	11
		medium	11
		loose	11
11	*How many spiral arms are there?*	1	05
		2	05
		3	05
		4	05
		more than four	05
		can't tell	05

Table 6.1: The 11 questions and corresponding 37 different possible answers for the Galaxy Zoo decision tree that users are taken through when classifying a galaxy.

since we distribute the 60% answering "features or disk" on the first question into two parts of relative size 70% and 30%. Additionally, the answers to Question 6 have been normalized to sum to 1.

6.1.1 Challenges with the data and interesting observations

Colors

An initial challenge with the data is the color in the images. Each image is represented by a $424 \times 424 \times 3$ tensor, where the third dimension represents the colors red, green and blue. This means that each image consists of 539,328 features. The three color-channels have a lot of information in common, and it is not clear whether the color contains any information, or whether converting the image to grayscale will preserve the information in the image.

Extracting image features

If we ignore the color aspect, each image sample still consists of 179,776 features. With this many features – many of which are highly correlated since pixels close to each other are highly dependent – the most important aspect is to construct a new set of good features.

6.2 Evaluation metric

The scoring method in this competition is Root Mean Squared Error defined as:

$$\text{RMSE} = \sqrt{\frac{1}{37N}\sum_{i=1}^{N}\sum_{j=1}^{37}(t_{ij} - p_{ij})^2},$$

where t_{ij} is the actual value for sample i, feature j and where p_{ij} is the predicted value for sample i, feature j.

6.3 Approach and progress

The content in this subsection is written chronologically to give an understanding of the decisions and insights we got while trying to solve this problem. When mentioning a submission to the leaderboard which achieves a best score so far, we will mark this score with bold.

We started out by making a simple benchmark submission. For each of the 37 outputs, we computed the average value in the training data, obtaining a vector of 37 elements. Submitting this vector as the prediction for each image in the test set gave a RMSE score of **0.16503**.

To make it easier to extract features and learn from the image data, we pre-processed the raw input images. The code for pre-processing is written in Python, using the scikit-image library. The pre-processing consisted of the following steps:

1. Threshold the image to black and white using Otsu's method – which is a method for automatically inferring the optimal thresholding level.
2. Use a morphological closing operation with a square of size 3×3 as kernel.
3. Remove all objects connected to the image border.
4. Remove all connected components for which the bounding box is less than 256 in area.
5. Find the connected component C closest to the center of the image.
6. Rescale the bounding box of C to be a square by expanding either the width or the height and add 15 pixels on each side.
7. In the original image, take out the portion contained in the expanded version of C.
8. Rescale the image to 64×64 pixels.

By performing the above steps, the images shown in Figure 6.1 are transformed to the images shown in Figure 6.3.

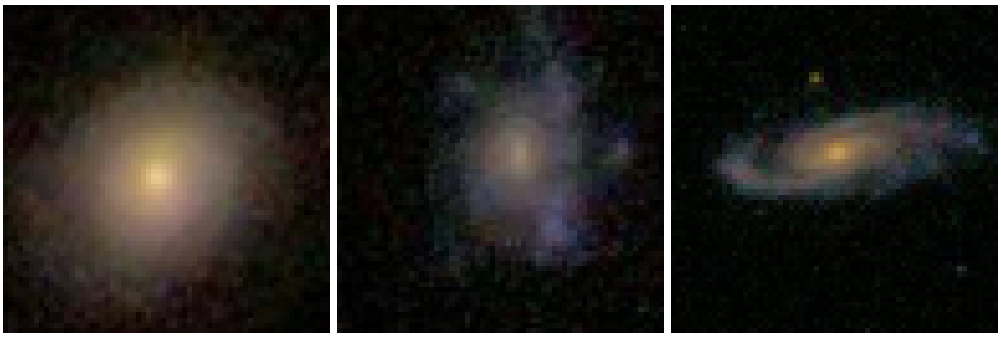


Figure 6.3: The images shown in Figure 6.1 after pre-processing. The original images were 424×424 pixels, where the processed images are 64×64 pixels.

Next we trained a convolutional neural network on the training data using the MATLAB implementation by [Palm, 2012]. The first layer in the network was a convolutional layer with 6 output maps and a kernel size of 21. The second layer was an average pooling layer with a scaling factor of 2. The third layer was a convolutional layer with 12 output maps and a kernel size of 5. The fourth layer was an average pooling layer with a scaling factor of 2. The final layer was a fully-connected layer with 37 output neurons. All layers used the sigmoid activation function. To avoid overfitting, we

trained the network with a validation set of 5,000 images. If the error on the validation does not lower over 5 epochs in a row, we stopped the training and saved the network with the minimum validation error.

Training this network architecture on 10,000 of the 61,578 training images, we obtained a leaderboard score of **0.13172**. Training the same architecture on 30,000 images from the training data, we obtained a leaderboard score of **0.12056**.

One way to get more performance out of a deep neural network, is to generate more training data using invariances in the data. One example of such an invariance in the galaxy data is rotational invariance since a galaxy can appear in an arbitrary rotation. Other possible invariances are translation and scale, but since the galaxies after pre-processing do not have much space on either side (they tend to fill out the image), this is not as easy to exploit. We trained the same network as before on 55,000 of the training images (this is Network 2 in Appendix B), where each image was also added after a rotation of 90 degrees, giving a total number of 110,000 training samples. Training the network took 80 epochs and a total of 53 hours.

When doing predictions on the test set, we sent each test image through the network four times, rotated 0, 90, 180 and 270 degrees, averaging the outputs. The validation error on the network was 0.1113. Feeding 4 different rotations through the network and averaging them gave a validation score of 0.1082. Finally, post-processing the output to obey the constraints of the decision tree reduced the validation error to 0.1077. On the leaderboard this gave a score of **0.10884**.

We extended the MATLAB library by [Palm, 2012] to have an additional fully-connected layer in the end of the network. Training with this extra layer did not improve the results of the network.

Next we tried a different architecture for the network (this is Network 8 in Appendix B):

1. Convolutional layer with 6 output maps and kernels of size 5×5
2. Factor 2 average pooling
3. Convolutional layer with 12 output maps and kernels of size 5×5
4. Factor 2 average pooling
5. Convolutional layer with 24 output maps and kernels of size 4×4
6. Factor 2 average pooling
7. Fully connected layer

This network reached a validation error of 0.1065 after 49 epochs (around 60 hours of training time). By inputting the test set with four different rotations and post-processing

the output, this gave a leaderboard score of **0.10449**.

We then tried the following architecture (this is Network 10 in Appendix B):

1. Convolutional layer with 8 output maps and kernels of size 9×9
2. Factor 2 average pooling
3. Convolutional layer with 16 output maps and kernels of size 5×5
4. Factor 2 average pooling
5. Convolutional layer with 32 output maps and kernels of size 5×5
6. Factor 2 average pooling
7. Convolutional layer with 64 output maps and kernels of size 4×4
8. Fully connected layer

This network reached a validation error of 0.0980 after 76 epochs (around 160 hours). On the leaderboard this network obtained a score of **0.09655** (with post-processing and rotated test images).

6.3.1 Visualizations of the trained networks

Where normal neural networks are fairly hard to interpret, it is easier with convolutional neural networks to understand what is happening. To understand how images are processed by the networks we have plotted some of the learnt kernels and visualizations of how an image is propagated through the network.

For these examples we have used Network 10 (see Appendix B for a description and comparison of the different networks trained) obtaining a leaderboard score of 0.09655. In Figures 6.4 and 6.5 we have plotted the 8 and 16 kernels from the first two convolutional layers (in the third and fourth convolutional layer the kernels are too small to interpret). By visual inspection it is not completely intuitive what convolution by these kernels is achieving.

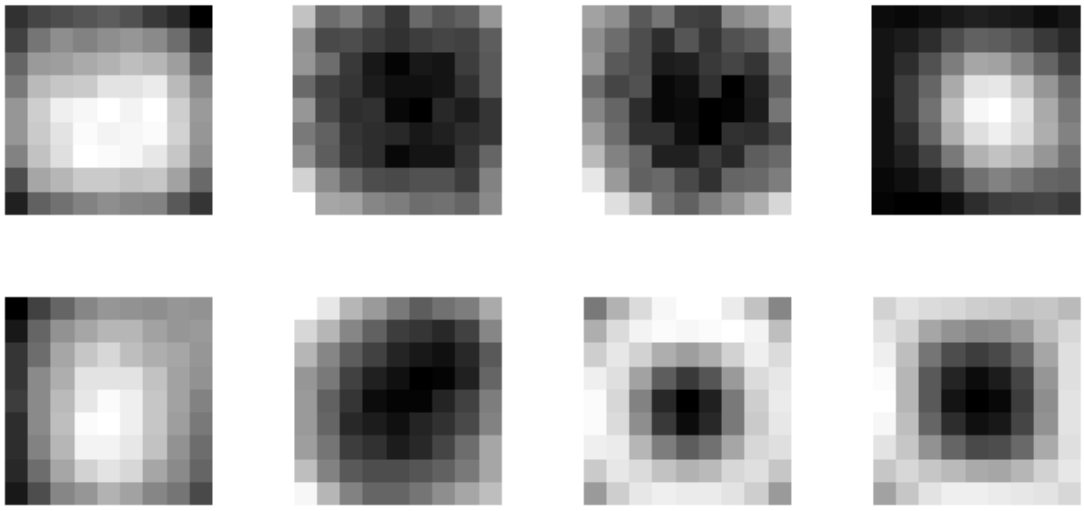


Figure 6.4: The 8 learnt kernels in the first convolutional layer of Network 10.

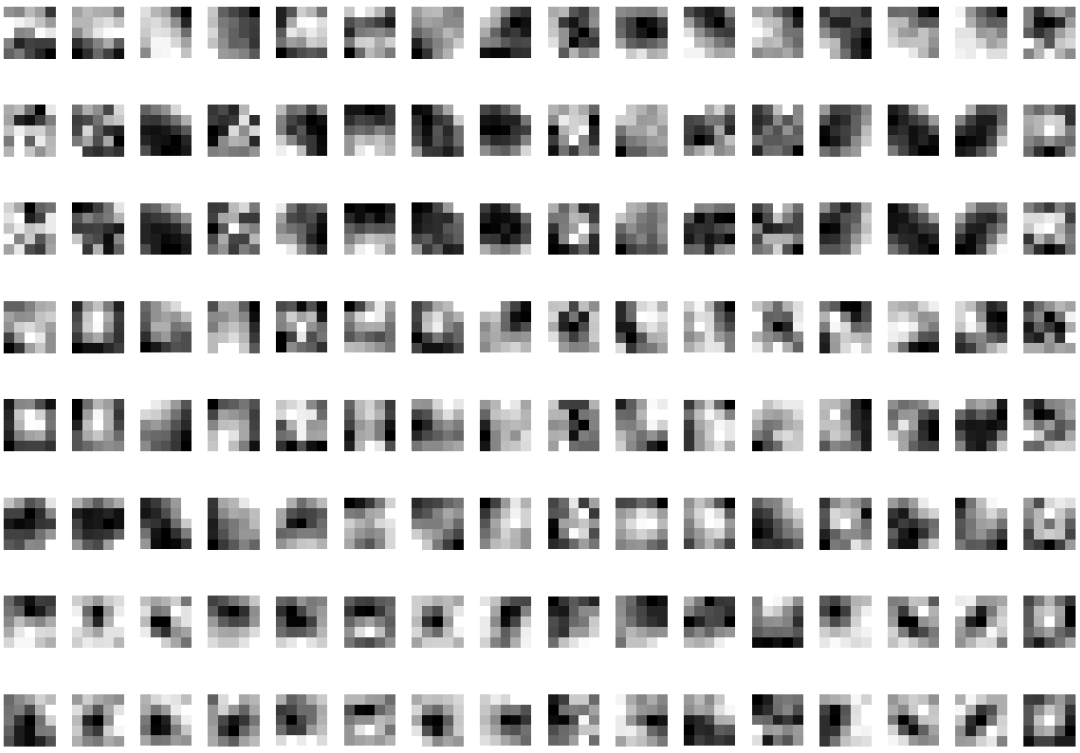


Figure 6.5: The 16 learnt kernels in the second convolutional layer of Network 10. Note that each of these 16 kernels is 3-dimensional and consequently each column of this figure corresponds to one kernel.

To get a better understanding of how the learnt kernels process the images we have taken two different galaxies and propagated them through the network, this is seen in Figures 6.6 and 6.7. On the top is the original image which is input into the network. In the next layer the first 8 kernels have been applied, in the next layer the next 16 kernels and so forth. Looking at the resulting output images after the first convolution gives a better idea of what the kernels are doing. Some of the kernels are extracting a very rough outline of the galaxy (kernel 2 and 3 in Figure 6.6). Some of the kernels are detecting the center of the galaxy (kernel 4 and 5 in Figure 6.6). Finally some of the kernels are detecting shapes/edges of some kind and consequently enhancing the shape of the galaxy (kernel 1,6,7 and 8 in Figure 6.6).

In the later layers of convolutions, the images are further broken down into differently shaped parts, extracting a reduced representation of the shape of the galaxy.

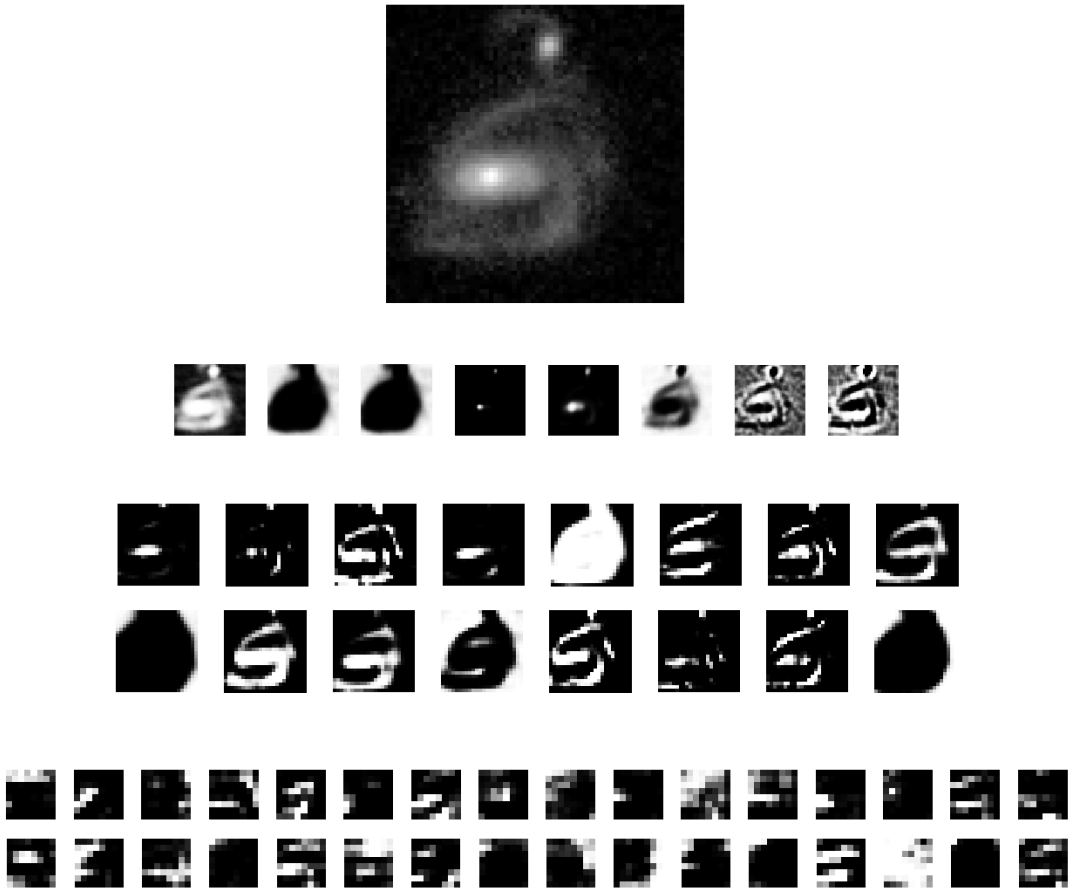


Figure 6.6: A visualization of how an image is propagated through the network. On the top is the original image which is input into the network. In the next layer the first 8 kernels have been applied, in the next layer the next 16 kernels and so forth.

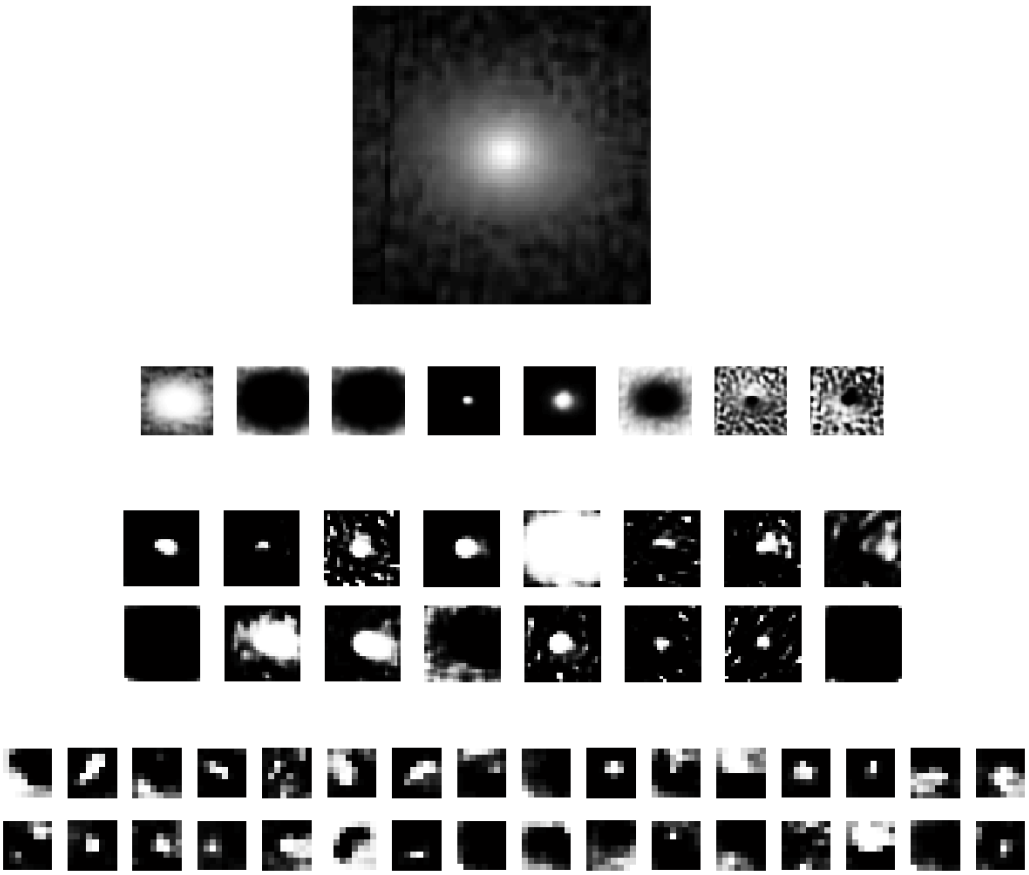


Figure 6.7: A visualization of how an image is propagated through the network. On the top is the original image which is input into the network. In the next layer the first 8 kernels have been applied, in the next layer the next 16 kernels and so forth.

To get a better understading of the weaknesses of our networks we have determined the 9 galaxies in the validation set which gives rise to the largest errors. These galaxies are shown in Figure 6.8. For some of them (the top left, the right in the center row, the bottom left and the middle one in the bottom row) it is easy to see what the issues are – either no galaxy is present in the image or multiple galaxies are present. For the two images where there is no galaxy present in the image the original images had large artifacts and noise.

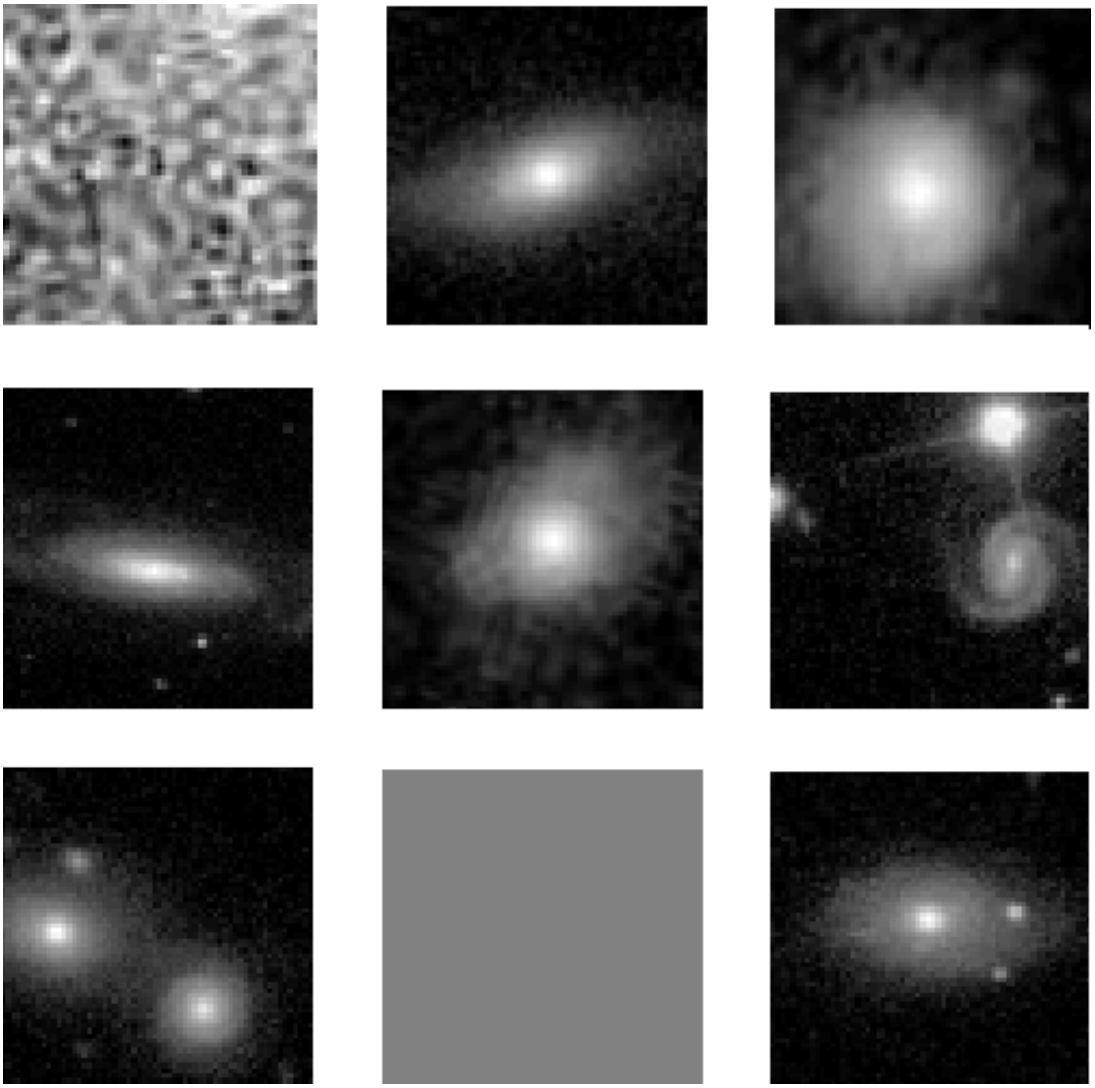


Figure 6.8: The 9 galaxies in the validation set which gives rise to the largest root mean squared error. For some of these images it is very clear why the network is struggling.

6.4 Results and discussion

Since the Galaxy Zoo competition is not over until the 4th of April 2014, this section is written without knowledge of the winning submission and without knowledge of the final standings on the leaderboard. When this thesis was handed in, our latest submission held a 15th position out of 223 possible.

6.4.1 Choice of model

Convolutional neural networks are complicated when it comes to model complexity. If we were to follow the hypothesis that simple models can get good results, starting out with convolutional nets would not be a wise decision. There are multiple reasons for going directly to convolutional nets: The first reason is that we have a very large dataset (more than 50,000 data samples). The second reason is that the data has obvious invariances (translation and rotation) which we can use to generate even more data. Finally, most recent litterature on image recognition points to convolutional nets as the best possible approach [Ciresan et al., 2012]. One possible alternative approach would be to use k nearest neighbors which would be very fast to implement. Unfortunately this approach (although simple) scales with the size of the dataset and consequently becomes relatively time-consuming.

Even though we do not know the inner workings of the other participants in the top of the leaderboard, we can still infer a bit of information from the forums and likewise. The person on the top of the leaderboard at this point, is Maxim Milakov, the author of nnForge – one of the best implementations of convolutional neural networks available – who has specialized in using convolutional neural networks for his submissions. Multiple threads on the Kaggle-forum have also indicated that deep neural networks is the solution participants are going for.

6.4.2 Speed of implementation

One significant issue was the speed of our implementation. When training the final convolutional nets the training time approached three weeks per network. In principle there was no limit to amount of data we could generate, and there were multiple ways to make the networks more complex. When training very complex models like convolutional neural networks, the efficiency of the implementation can be the deciding factor. There are other implementations of convolutional neural networks written in C^{++} which also uses the GPU for computations, this leading to large speed-ups.

6.4.3 Predicting the right thing

The main short-coming of our current approach is that we are not predicting exactly the correct thing. Regarding the error-measure we are doing the correct thing by optimizing mean squared error for the net, but when we are training the network we are treating the problem as a regression problem instead of the multi-class classification problem it really is. The problem with this is that the output is actually limited by 11 constraints which the networks are not utilizing (the answer-proportions for the 11 different questions in the decision tree must sum to specific values).

A better way to handle this problem is to construct a network for multi-class classification using a number of so-called *softmax units*, and to derive a correct loss-function for the network which incorporates the constraints of the decision tree directly.

Possible approach to modelling the correct loss function

The softmax function is a generalization of the logistic function which maps a vector of k elements to a new vector of k elements:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}}.$$

The elements of the vector $\sigma(x)$ are all between zero and one and they sum to one. The softmax function is often used as the last layer in a neural network for classification since the outputted values will resemble probabilities.

For our special classification task, we can model the 11 different questions using 11 distinct softmax units in the last layer. Now we can write the complete loss function in the following way (using mean squared error instead of root mean squared error for simplicity):

$$E = (t_{1a} - p_{1a})^2 + (t_{1b} - p_{1b})^2 + (t_{1c} - p_{1c})^2 + (t_{2a} - p_{2a}p_{1b})^2 + \dots,$$

where p_{1a} is our estimate of the proportion of users choosing answer a to question 1 and t_{1a} is the actual proportion. Note that we now have that for each question, the answer-proportions sum to 1 (as expected for a multi-class classification problem). The reason for multiplying p_{2a} and p_{1b} is that users only see question 2 if they answer b to question 1. We wrote a Python-script to generate the complete loss function, which takes up more than 60,000 characters if written in the following format:

```
(T1a - P1a)^2 + (T1b - P1b)^2 + (T1c - P1c)^2 + (T2a - P2a*P1b)^2 +
(T2b - P2b*P1b)^2 + (T3a - P3a*P2b*P1b)^2 + (T3b - P3b*P2b*P1b)^2 +
(T4a - P4a*P3a*P2b*P1b - P4a*P3b*P2b*P1b)^2 +
(T4b - P4b*P3a*P2b*P1b - P4b*P3b*P2b*P1b)^2 + ...
```

To avoid implementing this loss function directly, an alternative approach is to transform the target variables in an appropriate way. By using the description of the decision tree, we can transform the target values such that they sum to 1 for each of the 11 questions (thus making the "cumulative" probabilities into actual probabilities).

To motivate this idea, consider the following loss function:

$$E = (p_{1a} - t_{1a})^2 + (p_{1b} - t_{1b})^2 + (p_{1c} - t_{1c})^2 + (t_{1b}p_{2a} - t_{2a})^2 + (t_{1b}p_{2b} - t_{2b})^2 + \dots,$$

which can be rewritten to the form:

$$E = (p_{1a} - t_{1a})^2 + (p_{1b} - t_{1b})^2 + (p_{1c} - t_{1c})^2 + t_{1b}^2 \left(p_{2a} - \frac{t_{2a}}{t_{1b}}\right)^2 + t_{1b}^2 \left(p_{2b} - \frac{t_{2b}}{t_{1b}}\right)^2 + \dots$$

Here the fractions corresponds to new transformed target values, where we have rescaled the target variables such that the targets for each question will sum to one.

It is now possible to derive new gradients (and consequently δ's) for this new error function (where we have 11 softmax activation functions instead of the usual sigmoid activation functions in the last layer).

Unfortunately, due to time-constraints we did not have time to implement this in the convolutional neural network library.

Chapter 7

Conclusion

In this thesis we investigated competitive predictive machine learning, with a main focus on the platform provided by Kaggle. In Chapter 2 we stated a set of hypotheses about the approaches to be taken in such competitions, namely that:

1. Feature engineering is the most important part.
2. Simple models can take you very far.
3. Ensembling is a winning strategy.
4. Overfitting to the leaderboard is an issue.
5. Predicting the right thing is important.

We investigated these hypotheses using information about the winning entries in previous Kaggle competitions, by interviewing top participants from Kaggle and where possible by mathematical justification. Due to the diversity in the competitions at Kaggle combined with the breadth of data analysis approaches, nothing can be stated with absolute certainty about predictive modelling. Still the five rather general hypotheses seem to be supported by our investigations.

As an attempt to describe more applicable methodologies for predictive modelling, we outlined a conceptual framework in Chapter 3. This framework describes the process from data exploration, over data pre-processing to model training and finally selecting the right model. We describe common tricks and techniques for different types of problems and provide some insight into the minds of the best predictive modellers through qualitative interviews. Since no two datasets are the same, and since the field of machine learning is under constant development, it is not possible to state a perfect recipe for winning a competition at Kaggle. Still by utilizing the lessons learnt by previous competitors one can without too much previous experience obtain decent results.

To get a better understanding of the mechanics of Kaggle, to validate our hypotheses and to test out the conceptual framework we participated in two competitions at Kaggle. In the challenge posed by Expedia – described in Chapter 5 – the task was to rank hotels by their likelihood to be booked. In this competition we obtained a not very impressive ranking of 138th place out of 337 possible. Looking at the description of the winning solutions our approach was doing most of the things correctly, but a few important things were overlooked. Firstly we might have handled missing values in the wrong way by simply imputing the median value for many features. Secondly our methods did not take the order/position of the hotels into account which had a very clear effect. Finally one could reduce the training data dramatically with under-sampling while actually gaining better results. Most likely, these errors combined is the reason we ended up in 138th place – but it is also possible that some part of the result can be attributed to erroneous implementations by us.

In the challenge posed by Galaxy Zoo – described in Chapter 6 – the task was to classify galaxies by their shape given image data from deep space telescopes. Since this competition is not over yet, it is not possible to conclude completely on the results. Still we can say that our solution using deep neural networks is doing a fair job (at this time ranked 15 out of 223 on the leaderboard).

There is no doubt that competitive machine learning will be an important trend both in the academic community and for businesses in the future. Kaggle has provided a platform for running these competitions and so far the results obtained using the platform look very promising. Like for most other scientific challenges, no perfect algorithm exists for creating a winning solution on Kaggle since these often require creativity, patience and a bit of luck. Nevertheless, the methods and rules of thumb provided in this thesis can form a solid foundation for new participants in the future.

Appendix A

Previous Competitions

In this appendix we have summed up some details from 10 of the most recent competitions on Kaggle. We shortly describe the objective of the competitions and quote the description of the solutions by the Top-3 participants. Throughout the thesis, these competitions are referenced with the following abbreviated names: FACEBOOK, SUNNYHASHTAGS, SEECLICKPREDICT, BIRD, ACCELEROMETER, SOLARENERGY, STUMBLEUPON, BELKIN, BIGDATA and CAUSEEFFECT.

A.1 Facebook Recruiting III - Keyword Extraction

A.1.1 Objective

This competition tests your text skills on a large dataset from the Stack Exchange sites. The task is to predict the tags (a.k.a. keywords, topics, summaries), given only the question text and its title. The training set is large, consisting of millions of questions. The evaluation metric was Mean F1-Score – meaning that the competitors should predict a set of candidate tags, some which might not even be present in the training set.

A.1.2 Winning entries

1st: Owen

> My approach:
>
> 1. Take duplicates out like many others did
>
> 2. Calculate the association between all 1-gram and 2-grams in title and body. I kept the first 400 chars and last 100 of body to keep the size under control. I keep only the top 30 tags for each 1-gram and 2-gram
>
> 3. Predict the tags by combining all the tags associated with all the 1-gram and 2-grams in Title+body. Each 1/2 gram's impact is weighted by

its support and entropy. The entropy part only improve the score very marginally.

4. Take the top tags based on their score. The cut off threshold is determined by the ratio between the score of tag_k and the average score of all tags with higher scores than k.

5. I scored the above on 200k posts in training, and computed the bias for each predicted tag. Some tags have more FP than FN, some have more FN than FP. The goal is to have FP and FN about equal.

6. Score test data and adjust for the tag bias from step 5, i.e., if a tag has $\#FP > \#FN$, decrease its score, and vice versa.

Note: There are quite a number of tuning parameters that I "hand optimized" by evaluating the impact on 2% of training data I kept as a validation dataset. Title is much more important than body – I gave a 3:1 weight to title (but this is somewhat cancelled out by the fact that there are more words in body usually) I thought about doing some extraction but didn't get time to do so. I also extracted quoted posts in the body and tried to match that with training, just like the duplicates. But this barely improves the result. So it seems the association of tags between posts and quoted posts are not very strong. I also tried some TF-IDF based methods but didn't get good results. I didn't spend too much time on "NLP" like tasks such as stemming/tagging, etc. Part of the reason is time required, and part of reason is that the posts are not very "natural", compared to texts from news archives and books. I figure I probably would lose as much information on the code part, v.s. what I can gain in the "natural language" part.

The "RIG": As noted by everyone, the bottleneck of this competition is RAM. I am fortunate enough to have access to a machine with 256GB ram to try out different parameters.

The final solution would run on a desktop with 64GB of ram in about 4 hours. The training only takes about 1.5 hour and scoring will take 2.5 – partially because it runs out of ram. Training of the model will eat up almost all of the 64GB, and the rest will swap to disk. There has to be at least 90GB swap space for this to run with out error.

I used Python, which is not very "memory efficient". If we were to compete on "efficiency" I think c/c++ would be necessary.

2nd: Ruslan Mavlytov

This solution was not described anywhere.

3rd: E. G. Ortiz-García

This solution was not described anywhere.

A.2 Partly Sunny with a Chance of Hashtags

A.2.1 Objective

In this competition you are provided a set of tweets related to the weather. The challenge is to analyze the tweet and determine whether it has a positive, negative, or neutral sentiment, whether the weather occurred in the past, present, or future, and what sort of weather the tweet references.

A.2.2 Winning entries

1st: aseveryn

> As a core prediction model I used Ridge regression model (of course with CV to tune up alpha). I saw most of the people ended up using Ridge as well. I tried SGD and SVR but it was always worse than simple Ridge. My set of features included the basic tfidf of 1,2,3-grams and 3,5,6,7 ngrams. I used a CMU Ark Twitter dedicated tokenizer which is especially robust for processing tweets + it tags the words with part-of-speech tags which can be useful to derive additional features. Additionally, my base feature set included features derived from sentiment dictionaries that map each word to a positive/neutral/negative sentiment. I found this helped to predict S categories by quite a bit. Finally, with Ridge model I found that doing any feature selection was only hurting the performance, so I ended up keeping all of the features ~1.9 mil. The training time for a single model was still reasonable.
>
> Regarding the ML model, one core observation, which I guess prevented many people from entering into the $< .15$ zone, is that the problem here is multi-output. While the Ridge does handle the multi-output case it actually treats each variable independently. You could easily verify this by training an individual model for each of the variables and compare the results. You would see the same performance. So, the core idea is how to go about taking into account the correlations between the output variables. The approach I took was simple stacking, where you feed the output of a first level model and use it as features to the 2nd level model (of course you do it in a CV fashion).
>
> In my case, the first level predictor was plain Ridge. As a 2nd level I used Ensemble of Forests, which works great if you feed it with a small number of features – in this case only 24 output variables from the upstream Ridge model. Additionally, I plugged the geolacation features – binarized values of the state + their X,Y coordinates. Finally, I plugged the outputs from the TreeRegressor into the Ridge and re-learned. This simple approach gives a nice way to account for correlations between the outputs, which is essential to improve the model.

Finally, a simple must-not-to-forget step is to postprocess your results – clip to [0,1] range and do L1 normalization on S and W, s.t. the predictions sum to 1.

2nd: Tim Dettmers

I used deep neural networks with two hidden layers and rectified linear units. I trained them with stochastic gradient descent and Nesterov's accelerated gradient on GPUs and used dropout as regularization with which I could train a net in 30 minutes. One thing which I have not seen before but greatly improved my results was to use what is best denoted as "dropout decay", i.e. reducing dropout together with momentum at the end of training along with a linear decreasing learning rate - this seemed to keep hidden activities decoupled while at the same time introduced more information into the network which was quickly utilized and decreased the cross validation error by quite a bit.

I tried unsupervised pretraining with both restricted Boltzmann machines and several types of autoencoders, but I just could not get it working and I would be interested to hear from anyone for whom it worked.

Random sparse Gaussian weight initialization was much faster to train than random uniform square root initialization - and it might have also increased generalization, but I did not test this thoroughly.

Other than that I also used ridged regression and random forests regression. The later model performed rather poorly but made errors which were very different from my net and thus decreased the error in my ensemble.

I split my cross validation set into two and fitted least squares on one half to get the best ensemble.

3rd: Duffman & Pascal

At first a quick tfidf with RidgeRegressor and clipping to 0, 1 lower and upper bounds got me .152, but I found out after playing with the parameters that it was easy to fall into local optima since the columns each behaved wildly different. When I joined up with Pascal, we decided to try and do a more customized approach since trying to find a global optima for all columns just wasn't realistic.

Base Models: A quick summary is that we created models based on the combinations of char/word, tfidf/countvec, Ridge/SGDregressor/Linear-Regression, and S, W, K Block. So that's 2 * 2 * 3 * 3 for 36 models. The models parameters were optimized based on some custom code (written by the amazing Pascal!) to find the best parameters so we don't have to optimize 36 models individually. We cut out 12 models worth of features by simply putting it on a spreadsheet and doing some analyses to see if it

adds any variance or not. We probably could have used Recursive Feature Selection and we did eventually, but the RFE took a ridiculously long time for us to test so we decided to use a spreadsheet to get an idea of how much we should remove before we actually used it. The vast majority of our submissions was just playing around with the various models and how they fit into our final ensemble.

Ensemble: We did a stack with RidgeRegression at the end. The reason we did a stack rather than a weighted averaged blend was we wanted to have the other columns help predict each other. So K columns would help predict w columns and vice versa. It also allowed for optimization by column automatically rather than optimizing by all columns that might not be relevant to each other.

We also tried ExtraTreesRegressor ensemble and that gave us an amazing boost to our CV, but not to the LB which was a first. In our case, we made the right choice not to trust the CV since I was sure there were train/test set differences causing it.

For a final boost, we did several more stacks with different splits since the final stack was determined by 70/30 split and blended them. That way we can come close to using all the trianing data for the stack.

A.3 See Click Predict Fix

A.3.1 Objective

The purpose of this competitions is to quantify and predict how people will react to a specific 311 issue. What makes an issue urgent? What do citizens really care about? How much does location matter? The dataset for the competitions contains several hundred thousand 311 issues from four cities. Your model should predict, for each issue in the test set, the number of views, votes, and comments.

A.3.2 Winning entries

1st: J.A. Guerrero

My first submissions were based in naive dataset with features like:

- day(from 0 to 625)

- latitude

- longitude

- hour (beginning at 6:00 am)

- week_day (beginning on Monday)

- summary_len

- description_len

- city (factor)

- tag_type(onehot of categories with more than 20 cases)

- source(onehot of categories with more than 20 cases)

With this data and using R gbm I get results in 0.304x training the last 60 days. More training days always fit worst, and I realized optimizing the number of trees for different periods of training reflected in very different number of trees for reasonable values of learning rates. So for gain robustness I decided use small learning rates in all the models, 0.002 to 0.0005 for guarantee a more stable error curve.

Using absolute time feature (day) in a time series model has an obvious risk. You easily can learn time anomalies in training period but the extrapolation to the test period could be a lottery. Linear models project the data in a linear trend. Tree based models will assume the test period is like the last days of training period: always day feature is used in a subtree, the day values presents in test period are in the same branch that last training period.

In this case we were lucky so the test period means value were like the end of training period, but this could be different in a future. This fact leads me to open a new question: Was RMSE the best metric for this dataset? I personally think not, I see more interesting a ranking of issues than an exact estimation of number of votes and this would have prevented against calibration issues we've seen in this competition. In my definitive analysis I'll use Spearman rank coefficient too for model benchmarking.

But the metric was RMSE so my objective was build features time independents and use the full period of training.

The hypothesis were:

- The response to an issue depends (directly or inversely) of number of recent issues and similar issues (time dimension).

- The response to an issue depends (directly or inversely) of number of issues and similar issues reported close (geographic dimension).

- There are geographic zones more sensitive to some issues (geographic dimension).

With that in mind I defined three time windows, (short, middle and long) of 3, 14 and 60 days and three epsilon parameters (0.1, 0.4 and 1.2) for use them in a radial basis distance weighted average for each issue. The selection of this values were for adjust the decay shape in a way the weights represent city, district and neighbour ambits.

The tag_type were grouped in: crime_n_social, rain_snow, traffic, lights, trees, trash, hydrant, graffiti, pothole, NA, Other.

For each issue (row) I computed 3 (short, middle, long) x 3 (city, district and neighbour) features for each of 11 tag_group. Each feature uses radial basis weights respects the distance in kms between issues.

I computed 3 x 3 features for the total of issues and for the issues of the same group, so in total I had 3 x 3 x 13 of such features, all computed with a LOO (Leave One Out) criteria for avoid overfitting. This 117 feature set I named LOO features.

For a period of last 150 days and for each issue I computed the LOO weighted radial basis average for comments, votes and views for (city, district and neighbour) params (9 features) and the same but filtered to the issues in the same group (other 9 features). This 18 features I named BAYES features.

LOO and BAYES featured were normalized to (0,1) range for use with linear models too.

For summary I computed a bag of more frequents words ($>$ 50). Named BOW.

I fitted several models: (gbm, RF, glm) for (basic data, basic + LOO, basic + LOO + BAYES, basic + LOO + BAYES + BOW)

In the basic data added to other features I didn't used "day" feature forcing the model to learn the time anomalies from LOO features. I used longitude & latitude, but really the model learned from LOO very well too: longitude and latitude weren't neccesary.

Some models I fitted segmented by cities and almost all I used a "big column" approach for training the three responses all together. The models fitted with "big column" were systematically better than trained each response alone.

For the final blending I used the same method that James Petterson (BigChaos method of netflix prize) introducing some dummies models for each response and each city (all 0 except a city or a response).

Edit: And, of course, all done in log(x+1) scale and reversed to exp(x-1)

2nd: Bryan Gregory & Miroslaw Horbal

Overall I trusted CV for every decision I made for my model. My CV method was to chop off the final 20% of the training data (which worked out to be 44626 issues)

I treated scaling and selecting the number of issues to use at training time as a hyperparameter selection problem, so both my choices for scales and the number of training examples was selected from cross validation. I also used segmented scaling similar to Bryan, but my segments were broken down into:

Chicago, Chicago remote-api-created, Oakland, Oakland remote-api-created, New Haven, and Richmond

As Giovanni guessed, I focused a lot more on text-based features and TFIDF actually gave me the biggest single gain over any other features.

I trained a Ridge model on log(y + 1) targets and engineered the following features:

- TFIDF vectorization for summary and description up to trigrams

- boolean indicator for weekend

- log(# words in description + 1)

- city (one hot encoding)

- tag_type (one hot encoding)

- source (one hot encoding)

- time of day split into 6 4h segments (one hot encoding)

Along with those base features I also generated higher order combinations of some of the categorical features to produce new categorical one hot encoded features, these included:

- (city, time of day)

- (city, source)

- (city, tag_type)

- (source, tag_type)

Furthermore, I added 2 extra geographic features using data collected from a free geocoding service, these included:

- zipcode

- neighborhood name

and the combination:

- (zipcode, source)

Since there were a lot of sparse elements in my dataset I threshold any rare categories using various techniques. For tag_type, and higher order combinations I replaced any rare categories with a single "__rare__" category. For zipcode and neighborhoods I used a knn clustering heuristic I hacked together that essentially grouped rare zipcodes/neighborhoods with their nearest euclidean neighbour (lat, long) using an iterative process

Also, Bryan noticed that votes never drop below 1, so I was able to squeeze out a few extra points by setting 1 as a lower bound on votes.

Overall this model would score around 0.29528 on the private leaderboard. I think the main reason why Bryan's model and my model blended so well was primarily due to us independently coming up with very different,

equally powerful models that each had their own strengths. We gained 0.0035 on our score by applying a simple 50/50 weighted average as Bryan described.

3rd: James Petterson

This solution was not described on the forum.

A.4 Multi-label Bird Species Classification - NIPS 2013

A.4.1 Objective

The Neural Information Processing Scaled for Bioacoustics (NIPS4B) bird song competition asks participants to identify which of 87 sound classes of birds and their ecosystem are present in 1000 continuous wild recordings from different places in Provence, France. The training set contains 687 files. Each species is represented by nearly 10 training files (within various context / other species).

A.4.2 Winning entries

1st: Mario

Given in an external paper [Lasseck, 2013].

2nd: les bricoleurs

you can make a ~0.895 model by

- split each clip into half-second windows overlapping by 80%. for each window, determine the average value of each mfcc coefficient. (so you have 16 features for each window)

- use a random forest classifier with 87 yes/no outputs

- average the probabilities for each window to get the probability for the entire clip

A longer description is available on the forum of the competition.

3rd: Rafael

Now as regards my approach it is an RF in a multilabel setting (so called Binary Relevance). That's all.

selector = RandomForestClassifier(n_estimators=80, random_state=SEED)

selector.fit(train_x, train_y)

xTrain = selector.transform(train_x)

xTest = selector.transform(test_x)

model = RandomForestClassifier(n_estimators=250, criterion='entropy', max_depth=None, min_samples_split=4, min_samples_leaf=3, max_features='auto', bootstrap=True,oob_score=True, n_jobs=1,random_state=SEED, verbose=0)

classif = OneVsRestClassifier(model)

classif.fit(xTrain, train_y)

preds = classif.predict_proba(xTest)

train_y is the given 687x88 binary matrix of species (I include noise class and then remove it during submission).

train_x is column stacked features matrix from 3 different sources.

Source 1: Treat spectrogram as picture, extract ROI's, superimpose ROI's on the initial spectrogram and keep only the enhanced spectrum (see attached figs). Partition the picture in 16 equal horizontal bands and derive mean, std, median, kurtosis and 90% percentile. Tried several others (entropy, skewness etc). Did not work. So first features 5x16 = 80 per recording a 687x16 training matrix). Only with these you get about 86.5 on both leaderboards. Code to extract ROI's is available both in matlab and python.

Source 2: Make a bag of measurements for each recording by measuring properties of each ROI. Everything I could find in ndimage angle BW and duration of each ROI. These measurements are coded using 3 Kmeans of 75, 25 and 15 centres (resulting to 100 features per recording. So S2 is 687x100. HOGs did not work for me and the gain was only getting to a final about 88%.

Source3: The most powerful feature is the one introduced by Belunga in MLSP bird challenge continuing on Nick Kridler's approach on whales. It is based on deriving normalized cross-correlation of ROI's with the spectrum. This leads to a 687x33000 matrix. I did a variation here: I keep ROI's of train and test into sorted lists (frequency the key) and I calculate cross-correlation only if a ROI of training set fits a ROI of test set in terms of frequency range and duration. This leads to most correlations to be set to 0 without being calculated. This way the matrix is calculated very fast. Belunga's is a bit more accurate I admit but takes too much time for this competition (many days on an I7 16GB RAM).

Averaging over 10 models with different seeds unexpectedly gave a boost from 90.6 to 91.6 in the public ldb. The rest was by putting the last 15 lower probs per recording to a v.small costant.

A.5 Accelerometer Biometric Competition

A.5.1 Objective

Given is approximately 60 million unique samples of accelerometer data collected from 387 different devices. These are split into equal sets for training and test. Samples in

the training set are labeled with the unique device from which the data was collected. The test set is demarcated into 90k sequences of consecutive samples from one device. A file of test questions is provided in which you are asked to determine whether the accelerometer data came from the proposed device.

A.5.2 Winning entries

1st: gaddawin

> Well, my approach was similar to others:
>
> 1) find consecutive sequences and build chains. I used logistic regression over some simple features (time interval between sequences (only $<$1000ms), coords difference, etc) to find consecutive sequences. I spent much time trying to make chains longer and longer. But there are no sequences with duration $>$1 hour (it seems that they were deleted during data preparation). So, it is impossible to build very long chains.
>
> 2)It's easy to predict true device for long chains (except chais with 2 devices as Jose mentioned).
>
> 3)For sequences in "bad" chains i used random forest classifier with 5 features (sampling rate difference, time of day, etc)
>
> 4)It is possible for many devices to find sequences (and chains!) that are "consecutive" (day was changed during data preparation, but time of day remains the same) to last device record. Predicting device for these chains is very easy.

2nd: Solórzano

> As for methods, my top entry was a blend of 24 models. Some of them are conventional single-sequence models, both leak-based and non-leak-based. But the most accurate models (individually scoring up to 0.993) are based on lists of sequences (which I call sequence chains) where only the frequency of the professed device is considered, using a binomial distribution. A lot could be said about the best ways to build probable sequence chains, but I won't get into that.
>
> In some cases, the professed device IDs that label sequences in a chain are not informative. For example, a chain might have only 2 professed device IDs total. In such cases, it helps to apply conventional classifiers to the individual sequences of the chain, and combine the results.

3rd: Target Leak Model

This solution was not described anywhere.

A.6 AMS 2013-2014 Solar Energy Prediction Contest

A.6.1 Objective

The goal of this contest is to discover which statistical and machine learning techniques provide the best short term predictions of solar energy production. Contestants will predict the total daily incoming solar energy at 98 Oklahoma Mesonet sites, which will serve as "solar farms" for the contest. Input numerical weather prediction data for the contest comes from the NOAA/ESRL Global Ensemble Forecast System (GEFS) Reforecast Version 2. Data include all 11 ensemble members and the forecast timesteps 12, 15, 18, 21, and 24.

A.6.2 Winning entries

1st: Leustagos & Titericz

> Our approach to this problem didnt used much feature engineering. We used mostly raw features.
>
> Guidelines:
>
> We used 3 fold contiguous validation (folds with years 1994-1998, 1999-2003, 2004-2007)
>
> Our models used all features of forecast files without applying any preprocessing, so we took all 75 forecasts as features
>
> For each station, we used the 75 forecasts of 4 nearest mesos. So with this we had 75x4 such features.
>
> Besides those forecast features we had the following: month of the year, distance to each used meso, latitude difference to each meso. In total it was aproximately 320 features (with the forecast ones).
>
> We trained 11 models for this, one for each forecast member (the 11 independent forecasts given)
>
> We averaged those 11 models optmising MAE.
>
> We used pythons GradientBoostedRegressor fo this task.
>
> Thats it!

2nd: Odeillo

> I also used a similar approach:
>
> - first averaged the 11 forecast members (as Owen)
>
> - used the whole training set
>
> - used the 75 features (15 x 5 hours)
>
> - added Month + Elevation + Lat + Lon (nevertheless only Month gave interesting correlations)

- for each of the 98 mesonet I made a linear interpolation of the four nearest GEFS points (weighted by the distance). This was the step that really improved my score!

- dswrf and pwat clearly appeared to be the most important and I added derived features for them. For example dswrf(H3)- dswrf(H2)

- used gradient boosting techniques with a program written in C#

During the competition, the Admin added the elevations of the GEFS but I was unable to find any interesting correlations with them.

3rd: Owen

I used a similar approach:

* No feature engineering except for a linear combination of dswrf at the 5 different time points.

* Averaged the 11 forecast members as the first step

* Kept all 75 features

* Used GBM in R, with "laplace" distribution to optimize for MAE

* Built 2 GBMs, one based on the 75 features from the nears mesos, one based on the weighted average of the 75 features (inversely proportional to distance) of the nearest 4

* Similar to the winning team's approach, also included days of year, and long/lat in the model.

A.7 StumbleUpon Evergreen Classification Challenge

A.7.1 Objective

StumbleUpon is a user-curated web content discovery engine that recommends relevant, high quality pages and media to its users, based on their interests. While some pages they recommend, such as news articles or seasonal recipes, are only relevant for a short period of time, others maintain a timeless quality and can be recommended to users long after they are discovered. In other words, pages can either be classified as "ephemeral" or "evergreen". The mission is to build a classifier which will evaluate a large set of URLs and label them as either evergreen or ephemeral.

A.7.2 Winning entries

1st: fchollet

Features: raw Tf-idf + pca of the output of a tf-idf transform, for the following:

- url

- text (body)

- titles + "keywords" metatags (why this combination? mainly because some pages lack metatags, and the closest thing are the titles)

- text + url + metatags for "keywords" and "description" + "alt" values for img tags + possibly titles (which is pretty much the combination of all data I extracted from the pages –I might have eliminated titles from the mix after finding they did not impact results much due to redundancy edit: actually titles were indeed included here) So, for all these sets of terms, compute in each case a Tf-idf, then train a PCA on the output.

Then each resulting set of vectors will be classified with a Random Forest (PCA vectors) and a Logistic Regression (raw vectors). That's 4*2 arrays of predictions, which are then merged with a simple linear classifier. I also experimented with SGD and a few others, not sure if they were part of the code that generated the winning submission, I'll dig into it. edit: SGD was not used for the winning submission, only LR and RF

Anyway that's it: "classical" classifiers (LR & RF) run on the tf-idf (or tf-idf + pca if necessary) of terms extracted from urls, texts (body), titles, metatags, and alt img tags –then predictions merged with a linear classifier. Each classifier was tuned separately (grid search).

One more thing: it might be interesting for the StumbleUpon engineers to note that working *only* with the urls gets you 90% of the way there. So you can be extremely accurate without even having to fetch a page, and with very little computation. In production I'd use something like this.

2nd: Maarten Bosma

First let me say: Variance was crazy in this composition. My third place (private score 0.88817) scored only 0.88027 on the public leader board. It took quite some nerve to select it as my final submission.

I actually made another submission, which would have won had I selected it (Public: 0.88167, Private: 0.88915). I though the other result was more robust, even though that one had a higher CV score and lower variance. You can really image me banging my head on the table right now. I think, the lesson here is: if you are paranoid enough about overfitting you can trust your CV score.

My first key insight was: this competition is not about which pages are long lasting. This is first and foremost about what people find interesting. The main topics of interest is food (especially recipes). Other topics are that are mixed (but mostly evergreens) are health, lifestyle and excise. Seasonal recipes are actually mixed too. Supposedly funny videos/pics, technology, fashion, sports, sexy pictures are mostly ephemerals. Some sites are in languages other than English, these are mostly (but not all)

ephemerals. Sometimes the frontpage of a news site was in there (e.g. http://www.bbc.co.uk) → ephemeral.

My second key insight was: the features other than text are useless. I only used text features.

I used an html parser and to actually parse the html that was given. I then gave different weights to each tag (h1, meta-keywords, title, ...). I also used the given boilerplate and a boilerplate that I extracted myself with a tool called boilerpipe. I had more than 100 of these weighted sets. I then used raw (normalized) counts, stemming, tfidf, svd and lda to preprocess them. For each of those more than 300 sets I used logistic regression to get predictions, which I than combined into an ensemble.

I did not use n-grams, but should have.

I also tried to use meta feature: how many words, what kind of words (pos-tag distribution, common/uncommon, ratio in dictionary) .... The idea was that items with a good (easy to read) writing style are more likely to be evergreens. I got 0.83 out of these features alone, but they did not seem to add anything to the ensemble.

I also have a little bit of secret sauce. Something that I spend quite some time on. This eventually just added a little bit to the final ensemble, but maybe is something I will explore more in future competitions.

3rd: michaelp

This solution was not described anywhere.

A.8 Belkin Energy Disaggregation Competition

A.8.1 Objective

In this competition, the participants are given a dataset of readings from the Electromagnetic Interference (EMI) that most consumer electronic appliances produce to identify signatures. Examples of such signatures can be seen in Figure A.1.

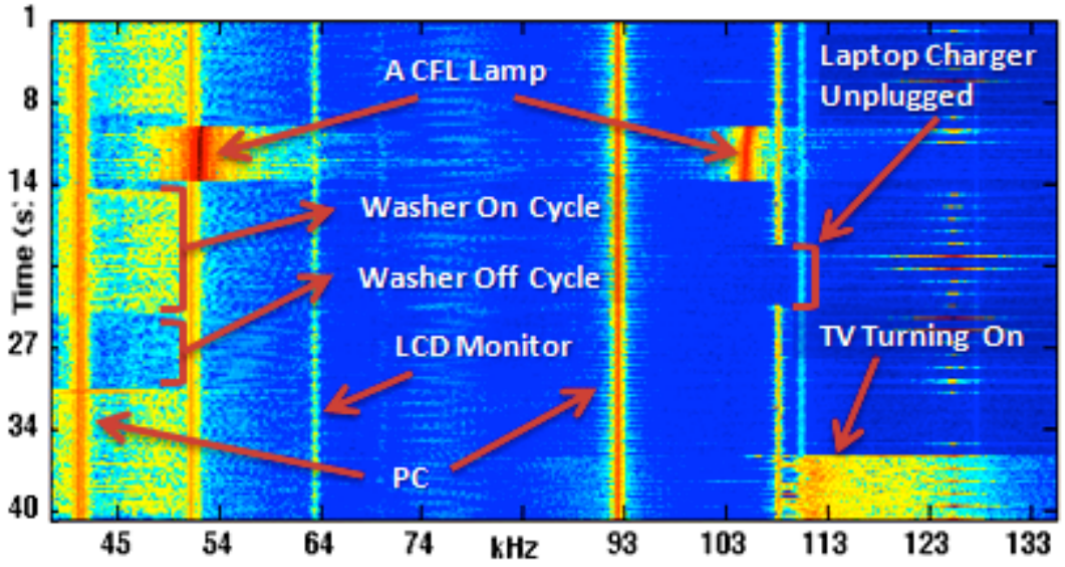


Figure A.1: Frequency spectrogram showing device actuation in a home. Taken from [Gupta et al., 2010].

A.8.2 Winning entries

1st: jessica bombaz

It was really a difficult task to detect the correct appliances. Although there are various data sources available, sometimes I found it impossible to distinguish or even detect low signal appliances due to high signal-to-noise ratio. This difficulty is also reflected in the final scores, as anyone of us managed to predict only about 0.04 / 0.08 = 50% of all the given appliances. I wonder, how much the score would be, if we took together the correctly predicted appliances of all the participants?

Basically I used also the first time differences to detect peaks. Instead of computing s(t+1) - s(t) which might be vulnerable to sampling frequency and also increases noise, my algorithms compute the mean of two time windows separated by some delay. Only then I took the difference. For those appliances with a characteristic signal, I computed the similarity by integrating over the squared difference. This works surprisingly well even if the signal is noisy.

2nd: Luis

My approach is still similar to the edge detection and differences in the power, but I will still describe it.

First at all, I did not use any information from the HF data. I did not even look at it. My plan was originally to extract as much information from the power as I can and then go back to use the HF information. Later, I did not have time to go back and explore the HF data.

For each of the appliances I have a window of 12 time ticks (2 seconds) that contain the "turning on" signature and another window for the "turning off" signature. The first 3 time ticks contain the power when the appliance is still off, and the remaining contain the power when the appliance is on. The number 12 is not fixed. I use different numbers for some appliances. There is a trade off for that number. For large numbers, the system is more robust to noise. For low numbers, the system is more capable of identifying appliances that are turned on or off at the same time.

For a new test day, I have a window that runs through each time tick of the day and is compared to the "turning on" signatures of each appliance. When the running window is very similar to one of the appliances signatures, the system mark that the appliance has been turned on. The same is done independently for the "turning off" signatures. Finally, I pair the "turning on" events with the "turning off" events. That is the basic algorithm.

My system is capable of identify appliances that are turned on very close in time (the difference in time should still be at least 1 second). However, it has limitations for low power appliances and it is unable to successfully distinguish appliances that are very similar.

3rd: Titan

I also used time differences (first derivative) to detect occurrence of events which as Rosnfeld already mentioned is critical. I used an edge detection algorithm for the power signatures to detect events and measure the size of the jump in each value from before and after the event happened. A summary of the results is shown in our visualization entry.

I could see few transient HF signals that lasted more than a second and even for those the one second resolution of the HF data was too coarse to be useful because in some cases the transient was concentrated in one second giving a clear signal and in other cases it was split between two consecutive seconds resulting in a weaker signal for each of the seconds that was hard to distinguish from the noise.

For each event, I averaged the HF data for several samples after the event and subtracted the average from several samples before the event. I then averaged these HF difference over multiple events to get steady state HF signatures for each appliance (plotted in the HFdiff figures of the visualization entry). In theory, these HF signatures should have allowed me to distinguish between appliances that had nearly identical power signatures and the theory worked well in some cases.

These were some appliance pairs for which even the HF signatured I generated were not clear enough to distinguish between them and other pairs where the back-end solution did not seem to match what the HF signatures were telling me. In some cases it was more effective to choose the laundry room lights based on whether the Washer/Dryer were working at the same time than to rely on the HF signatures.

A.9 The Big Data Combine Engineered by BattleFin

A.9.1 Objective

The first stage of the competition is a predictive modelling competition that requires participants to develop a model that predicts stock price movements using sentiment data provided by RavenPack. You are asked to predict the percentage change in a financial instrument at a time 2 hours in the future. The data represents features of various financial securities (198 in total) recorded at 5-minute intervals throughout a trading day. To discourage cheating, the features' names or the specific dates are not provided.

A.9.2 Winning entries

1st: BreakfastPirate

A high-level description of my approach is:

1. Group securities into groups according to price movement correlation.

2. For each security group, use I146 to build a "decision stump" (a 1-split decision tree with 2 leaf nodes).

3. For each leaf node, build a model of the form Prediction = m * Last Observed Value. For each leaf node, find m that minimizes MAE. Rows that most-improved or most-hurt MAE with respect to m=1.0 were not included.

2nd: SY

My model was designed with one main principle in mind: to make model as simple as possible and as stable as possible to avoid overfitting. My final submission was a weighted mean of three very similar models. Blending, however, generated very small gain in this particular case. That is why here I will be describing single model that was a part of the blend. This model by itself has 0.40959 score on public leaderboard and 0.42378 on private, which is enough to be # 5 on public and # 3 on private leaderboards.

An extremely long model description is not presented here, but can be found on the forums.

3rd: Ed Ramsden

> Most of the submissions I built relied on relatively simple linear models. Some more complex models I tried fit better and cross-validated better, but ranged from poor to truly horrendous on the public LB. Although the linear-type models did not perform all that well in either x-validation or public LB (or even for basic fit), they performed consistently.

A.10 Cause-effect pairs

A.10.1 Objective

As is known, "correlation does not mean causation". More generally, observing a statistical dependency between A and B does not imply that A causes B or that B causes A; A and B could be consequences of a common cause. But, is it possible to determine from the joint observation of samples of two variables A and B that A should be a cause of B? There are new algorithms that have appeared in the literature in the past few years that tackle this problem. This challenge is an opportunity to evaluate them and propose new techniques to improve on them.

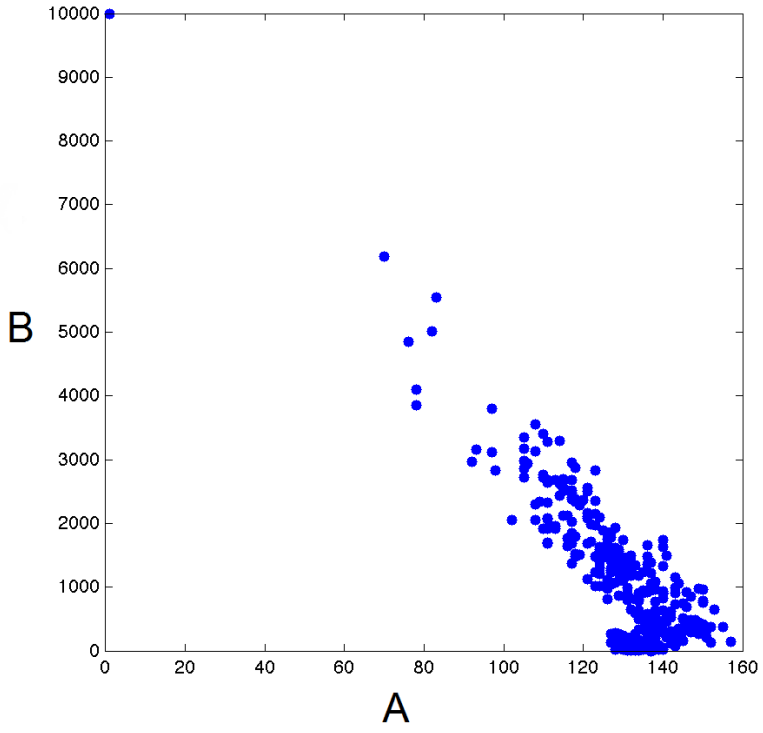


Figure A.2: In this plot, A is temperature and B is altitude of German cities, hence B causes A.

A.10.2 Winning entries

1st: ProtoML

What I think I did well:

1. Created lots of features. My final model, before feature selection, had almost 9000.

2. Compared Apples to Apples. I made sure that comparisons between input types (numerical, categorical, binary) were consistent.

3. Handmade features. Made a couple of features to proxy a "vertical line test" (essentially local variance).

4. Feature Selection. I made a genetic algorithm to do feature selection. This improved performance, though it wasn't necessary since my submission without feature selection still scored 0.817.

What I didn't do (that I probably should have):

1. Use any of the previous research. I read in the forum about all of these models in another thread (ANM, PNL, LINGAM, IGCI, etc.) when the competition was almost over, and I didn't want to bother including and that probably could have helped a lot.

2. Use more of the public info. I didn't use the four-way division at all, though I could have probably extracted more features out of it.

3. Create more features and ensemble. I was confident that doing this could have improved my score, but I was too distracted working on previously mentioned library to do so. This almost cost me the competition, hence my score plateauing in the end.

4. Test the impact of features that I added and made more that are similar. I'm unsure if this would be optimal. I feel like this should be done automatically, but since I don't have the ability to do so (yet), it probably could have helped tune the features more.

2nd: jarfo

This solution was not described anywhere.

3rd: HiDLoN

This solution was not described anywhere.

Appendix B

Trained Convolutional Networks

Here we give a brief description of the different networks we have trained as a part of the Galaxy Zoo competition (which is described in Chapter 6).

To describe the architecture of the networks we will use the following abbreviations: xCy will denote a convolutional layer with x kernels of size $y \times y$. The abbreviation Sz will denote an average-pooling layer with a scaling factor of z.

Name	Architecture	Dataset	Validation	Leaderboard
Network 1	6C21-S2-12C5-S2	1	0,1168	
Network 2	6C21-S2-12C5-S2	2	0,1113	0,10884
Network 3	12C21-S2-12C5-S2	1	0,1148	
Network 4	6C21-S2-12C5-S2	3	0,1152	
Network 5	6C21-S2-12C5-S2	3	0,1114	
Network 6	6C21-S2-12C5-S2	3	0,1115	
Network 7	6C21-S2-24C5-S2	3	0,1087	
Network 8	6C5-S2-12C5-S2-24C4	3	0,1065	0,10449
Network 9	6C5-S2-12C5-S2-24C4-S2-48C4-S2	3	0,0997	
Network 10	8C9-S2-16C5-S2-32C5-S2-64C4	3	0,0980	0,09655
Network 11	6C5-S2-12C5-S2-24C4-S2-48C4-S2	4	0,1021	
Network 12	8C9-S2-16C5-S2-32C5-S2-64C4	4	*0,0989*	
Network 13	6C5-S2-12C5-S2-24C4	4	0,1058	
Network 14	6C5-S2-12C5-S2-24C4-S2-48C4-S2	5	0,0981	
Network 15	8C9-S2-16C5-S2-32C5-S2-64C4	5	*0,0974*	
Network 16	6C5-S2-12C5-S2-24C4	5	0,1014	

Table B.1: A table giving a short description of the differences of the trained convolutional networks. For each network we show the architecture, which dataset was used, the obtained validation score and the score on the leaderboard (after post-processing). Network 12 and Network 15 did not finish training before handing in this thesis.

The datasets Throughout the competition we changed the data we fed to the networks multiple times. At first we used the pre-processed images (as described in

Section 6.3) in grayscale. We then added the same images but rotated 90 degrees, consequently doubling the size of the dataset. For Dataset 3 we also added rotations of 180 and 270 degrees giving a dataset with 220,000 images for training. To investigate the effect of pre-processing, Dataset 4 is the same as Dataset 3 except that the images are not pre-processed, . Finally we changed the network to handle color-images which is then tested with Dataset 5 which is a version of Dataset 3 with RGB-colors.

Name	**Pre-processing**	**Rotations**	**Colors**	**# Images**
Dataset 1	Yes		No	55,000
Dataset 2	Yes	90	No	110,000
Dataset 3	Yes	90,180,270	No	220,000
Dataset 4	No	90,180,270	No	220,000
Dataset 5	Yes	90,180,270	Yes	220,000

Table B.2: A table showing the differences between the different datasets used for training convolutional neural networks.

Training curves In Figure B.1 we have plotted the validation error during training for a subset of the networks described above. We have chosen the networks which were significantly better than earlier networks and which were consequently submitted to the leaderboard. Note that the time per epoch was not the same for these networks.

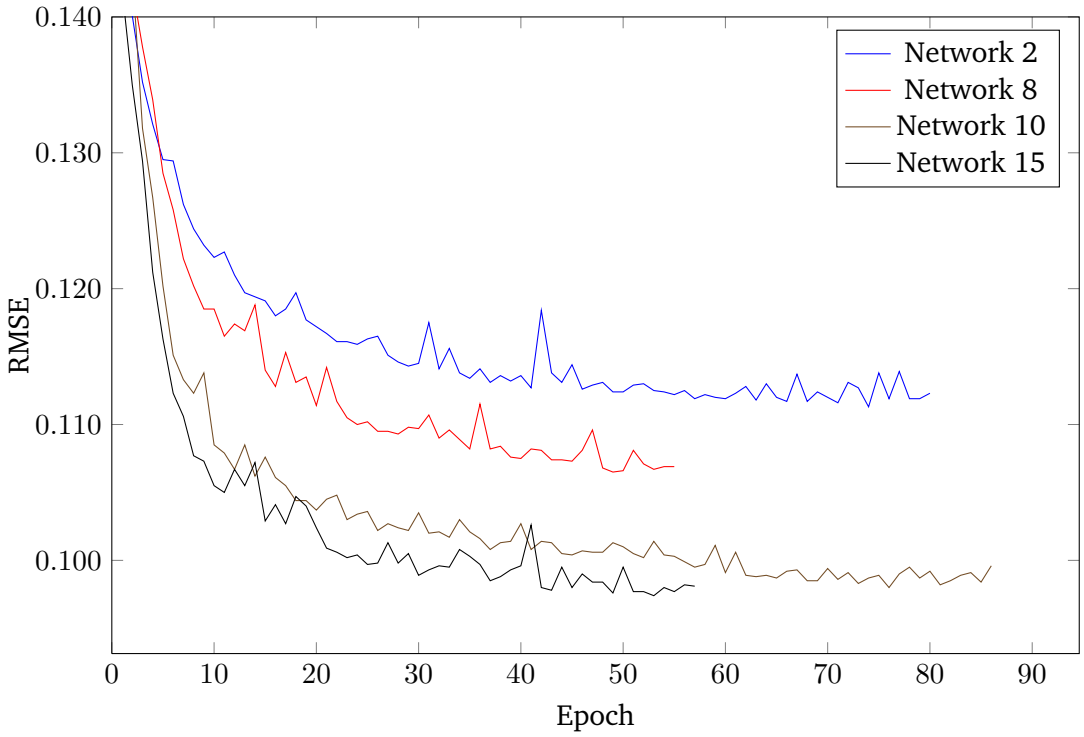


Figure B.1: Validation error during training for a subset of the networks described above.

Appendix C

Interviews with Kaggle Masters

To get a more qualitative idea about how to win a competition at Kaggle, we e-mail-interviewed three of the top Kaggle competitors.

C.1 Participants

C.1.1 Tim Salimans (currently ranked 40th on Kaggle)

PhD data scientist with expertise in predictive modelling and analytics. Co-founder of predictive analytics consulting firm Algoritmica. Winner of several predictive modelling competitions and a top lecturer award.

*(*Tim Salimans*)*

C.1.2 Steve Donoho - BreakfastPirate (currently ranked 4th on Kaggle)

He is founder and Chief data Scientist at Donoho Analytics Inc. Prior to this, he worked as Head of Research for Mantas and as Principal for SRA International Inc. On the education front, Steve completed his graduation from Purdue University followed by a M.S. And Ph.D. From Illinois University. His interest and work include an interesting mix of problems in areas of Insider trading, Money Laundering, Excessive mark up and customer attrition.

*(*Interview at `http://www.analyticsvidhya.com/`*)*

C.1.3 Anil Thomas (currently ranked 6th on Kaggle)

I am a Technical Leader at Cisco Systems, where I work on building multimedia server software. I was introduced to machine learning when I participated in the Netflix Prize competition. Other than Netflix Prize where I was able to eke out an improvement of 7% in recommendation accuracy, I have no significant data mining experience to speak of. (Anil Thomas)

C.2 Questions

Question 1 Predictive modelling combines many techniques - such as statistics, visualization, programming, intuition, creativity and handling data. Of the many skills in predictive modelling (especially in the competitive framework), which are the most/least essential? Which take up the most time during a Kaggle competition? Which do beginners usually lack?

Question 2 Following up on the first question, how important is a solid mathematics understanding of statistics and machine learning?

Question 3 When trying to win a Kaggle competition, what is your normal workflow? How do you initially look at the data? How much initial work do you put in building a solid pipeline (for example for cross validation)? Which models do you initially try? How much research into modern method do you do - if any, how? How does your work change during the competition?

Question 4 What have you learned from Kaggle competitions that has become valuable later? Do you in general have any memories of things that surprised you during/after a competition which made you change your workflow?

Question 5 How should one optimally use the leaderboard? Is is possible to "game the system" using the leaderboard?

Question 6 How is your take on the trend that most winning entries in the competitions are based on heavy use of ensembling?

C.3 Answers

C.3.1 Tim Salimans

Question 1 For most Kaggle competitions the most important part is feature engineering, which is pretty easy to learn how to do. Another important part is model combination (e.g. stacking), which might take a little bit longer. However, personally I prefer to only enter those competitions that require strong modelling skills and knowledge of statistics. This is an area many other competitors are lacking, which makes it easier to win.

Question 2 Very important for some (like the chess rating challenge, and observing dark worlds competitions I won), not so much for others (like Amazon's competition where I ended up 4th). Like I said, I try to select for the former type of competition.

Question 3 It is best to iterate quickly, so first build a very simple model and then expand upon that: In doing so I go through multiple cycles of feature engineering, model building, cross validation and model combination. Building the pipeline is definitely important: easy to to for standard competitions, but more difficult for non-standard ones. The types of models I use depend on the competition. I keep up to date on work in Machine Learning and statistics, but otherwise I don;t do a lot of research particular to a competition.

Question 4 I learned about the importance of iterating quickly. And trying out many different thinks and validating, rather than trying to guess th best solution beforehand. Also the whole model training $\rightarrow$ validation $\rightarrow$ combination workflow is something I picked up via Kaggle, but not sure how useful this is in other contexts.

Question 5 The leaderboard definitely contains information. Especially when the leaderboard has data from a different time period than the training data (such as with the heritage health prize). You can use this information to do model selection and hyperparameter tuning. Not really gaming the system I guess though, since the split between private and public leaderboard data makes it impossible to really use the final test data itself.

Question 6 This is natural from a competitors perspective, but potentially very hurtful for Kaggle/its clients: a solution consisting of an ensemble of 1000 black box models does not give any insight and will be extremely difficult to reproduce. This will not translate to real business value for the comp organizers. Also I myself think it is more fun to enter competitions where you actually have to think about your model, rather than just combining a bunch of standard ones. In the chess rating, don't overfit, and dark worlds competitions, for example, I used only a single model.

C.3.2 Steve Donoho

Question 1 The way I view it is that knowledge of statistics & machine learning is a necessary foundation. Without that foundation, a participant will not do very well. BUT what differentiates the top 10 in a contest from the rest of the pack is their creativity and intuition. Depending on the competition, data preparation can take up a lot of time. For some contests, the data is fairly simple – one flat table – and a user can start applying learning algorithms right away. For other contests – the GE Flightquest contests are an example – there is a lot of data prep work before one can even begin to apply learning algorithms. I think beginners sometimes just start to "throw" algorithms at a problem without first getting to know the data. I also think that beginners sometimes also go too-complex-too-soon. There is a view among some people that you are smarter if you

create something really complex. I prefer to try out simpler. I *try* to follow Albert Einstein's advice when he said, "Any intelligent fool can make things bigger and more complex. It takes a touch of genius – and a lot of courage – to move in the opposite direction." See my answer to #3 below for more info.

Question 2 The more tools you have in your toolbox, the better prepared you are to solve a problem. If I only have a hammer in my toolbox, and you have a toolbox full of tools, you are probably going to build a better house than I am. Having said that, some people have a lot of tools in their toolbox, but they don't know *when* to use *which* tool. I think knowing when to use which tool is very important. Some people get a bunch of tools in their toolbox, but then they just start randomly throwing a bunch of tools at their problem without asking, "Which tool is best suited for this problem?"

Question 3 Well, I start by simply familiarizing myself with the data. I plot histograms and scatter plots of the various variables and see how they are correlated with the dependent variable. I sometimes run an algorithm like GBM or randomForest on all the variables simply to get a ranking of variable importance. I usually start very simple and work my way toward more complex if necessary. My first few submissions are usually just "baseline" submissions of extremely simple models – like "guess the average" or "guess the average segmented by variable X." These are simply to establish what is possible with very simple models. You'd be surprised that you can sometimes come very close to the score of someone doing something very complex by just using a simple model.

A next step is to ask, "What should I actually be predicting?" This is an important step that is often missed by many – they just throw the raw dependent variable into their favorite algorithm and hope for the best. But sometimes you want to create a derived dependent variable. I'll use the GE Flightquest as an example – you don't want to predict the actual time the airplane will land; you want to predict the length of the flight; and maybe the best way to do that is to use that ratio of how long the flight actually was to how long it was originally estimate to be and then multiply that times the original estimate.

Cross-validation is very important. Once I have a good feel for the data, I set up a framework for testing. Sometimes it is cross validation. If it is time-related data where you are using earlier data to predict later data (SeeClickFix is a good example), it is better to split your training data into "earlier" and "later" and use the earlier to predict the later. This is more realistic compared to the actually testing.

As for research into modern methods – some contests force me to learn what the latest techniques are. Sometimes that will give you a small edge in a contest.

Question 4 I sometimes use Kaggle contests to force me to learn new techniques. For example, I didn't have much experience in text mining a year ago. So I have deliberately

entered a few text mining contests to force myself to learn the latest techniques. Kaggle makes the dangers of overfit painfully real. There is nothing quite like moving from a good rank on the public leaderboard to a bad rank on the private leaderboard to teach a person to be extra, extra careful to not overfit.

Question 5 The public leaderboard is some help, but as stated above, one needs to be careful to not overfit to it especially on small datasets. Some masters I have talked to pick their final submission based on a weighted average of their leaderboard score and their CV score (weighted by data size). I suppose there might be small ways to game the leaderboard, but for me it is simply more fun to try to come up with a good model.

Question 6 I am a big believer in ensembles. They do improve accuracy. BUT I usually do that as a very last step. I usually try to squeeze all that I can out of creating derived variables and using individual algorithms. After I feel like I have done all that I can on that front, I try out ensembles.

C.3.3 Anil Thomas

Question 1 That is a very good enumeration of required skills. When I started competing, my skills were pretty much limited to programming and creativity. I was a bit surprised to find out how far those skills alone can take you. There is no question that knowledge of statistics (also calculus and linear algebra) is a valuable asset in this field. Intuition gets better with exposure to a variety of problems and datasets. Creativity, I think buys you the most bang for the buck. There are times when an unusual/innovative approach gets you great results with little effort. I would add perseverance to the list, though it may be more of a trait than a skill.

Question 5 Having a good cross validation system by and large makes it unnecessary to use feedback from the leaderboard. It also helps to avoid the trap of overfitting to the public leaderboard. The organizers are usually careful to construct the problem in such a way as to prevent gaming (via the leaderboard or otherwise). There have been instances of data leakage that makes gaming possible. Contestants are usually quick to point these out.

Question 6 No matter how faithful and well tuned your individual models are, you are likely to improve the accuracy with ensembling. Ensembling works best when the individual models are less correlated. Throwing a multitude of mediocre models into a blender can be counterproductive. Combining a few well constructed models is likely to work better. Having said that, it is also possible to overtune an individual model to the detriment of the overall result. The tricky part is finding the right balance.

Bibliography

[Amit and Y, 1997] Amit, Y. and Y, D. G. (1997). Shape quantization and recognition with randomized trees. *Neural Computation*, 9:1545–1588.

[Bishop, 1995] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Breiman, 2001a] Breiman, L. (2001a). Random forests. *Machine Learning*, 45(1):5–32.

[Breiman, 2001b] Breiman, L. (2001b). Statistical modeling: The two cultures. *Statistical Science*.

[Burges, 2010] Burges, C. J. C. (2010). From RankNet to LambdaRank to LambdaMART: An overview. Technical report, Microsoft Research.

[Chapelle et al., 2011] Chapelle, O., Chang, Y., and Liu, T.-Y., editors (2011). *Proceedings of the Yahoo! Learning to Rank Challenge, held at ICML 2010, Haifa, Israel, June 25, 2010*, volume 14 of *JMLR Proceedings*. JMLR.org.

[Ciresan et al., 2012] Ciresan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649.

[Dang, 2011] Dang, V. (2011). RankLib. `http://www.cs.umass.edu/~vdang/ranklib.html`.

[Dasu and Johnson, 2003] Dasu, T. and Johnson, T. (2003). *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc.

[Deng et al., 2011] Deng, H., Runger, G., and Tuv, E. (2011). Bias of importance measures for multi-valued attributes and solutions. In *Proceedings of the 21st International Conference on Artificial Neural Networks - Volume Part II*, ICANN'11, pages 293–300, Berlin, Heidelberg. Springer-Verlag.

[Forum, 2012] Forum, W. E. (2012). Big data, big impact: New possibilities for international development. `http://www.weforum.org/reports/big-data-big-impact-new-possibilities-international-development`. [Online].

[Friedman, 2000] Friedman, J. H. (2000). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232.

[Goldbloom, 2010] Goldbloom, A. (2010). Data prediction competitions – far more than just a bit of fun. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 1385–1386.

[Gupta et al., 2010] Gupta, S., Reynolds, M. S., and Patel, S. N. (2010). Electrisense: Single-point sensing using emi for electrical event detection and classification in the home. In *In Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, pages 139–148.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

[Ho, 1995] Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, pages 278–, Washington, DC, USA. IEEE Computer Society.

[Ho, 1998] Ho, T. K. (1998). The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844.

[Hoyer, 2002] Hoyer, P. O. (2002). Non-negative sparse coding. In *IN NEURAL NETWORKS FOR SIGNAL PROCESSING XII (PROC. IEEE WORKSHOP ON NEURAL NETWORKS FOR SIGNAL PROCESSING*, pages 557–565.

[Lasseck, 2013] Lasseck, M. (2013). Bird song classification in field recordings: Winning solution for nips4b 2013 competition.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[Lee and Seung, 1999] Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by nonnegative matrix factorization. *Nature*, 401:788–791.

[Lee and Seung, 2000] Lee, D. D. and Seung, H. S. (2000). Algorithms for non-negative matrix factorization. In *In NIPS*, pages 556–562. MIT Press.

[Leek, 2013] Leek, J. (2013). Data munging basics (slides).

[Lohr, 2012] Lohr, S. (2012). The age of big data. `http://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html`. [Online; posted 11-February-2012].

[Manyika et al., 2011] Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., and Byers, A. H. (2011). Big data: The next frontier for innovation, competition and productivity. `http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation`. [Online; posted May-2011].

[Matsugu et al., 2003] Matsugu, M., Mori, K., Mitari, Y., and Kaneda, Y. (2003). Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, 16(5–6):555 – 559.

[McCullagh and Nelder, 1989] McCullagh, P. and Nelder, J. A. (1989). *Generalized linear models (Second edition)*. London: Chapman & Hall.

[Mu, 2011] Mu, Z. (2011). The impact of prediction contests.

[Palm, 2012] Palm, R. B. (2012). Prediction as a candidate for learning deep hierarchical models of data. Master's thesis, Technical University of Denmark.

[Rencher and Pun, 1980] Rencher, A. C. and Pun, F. C. (1980). Inflation of r2 in best subset regression. *Technometrics*, 22(1):49–53.

[Tumer and Ghosh, 1996] Tumer, K. and Ghosh, J. (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(3-4):385–403.

[Wickham, 2012] Wickham, H. (2012). Tidy data.

[Willett et al., 2013] Willett, K. W., Lintott, C. J., Bamford, S. P., Masters, K. L., Simmons, B. D., Casteels, K. R. V., Edmondson, E. M., Fortson, L. F., Kaviraj, S., Keel, W. C., Melvin, T., Nichol, R. C., Raddick, M. J., Schawinski, K., Simpson, R. J., Skibba, R. A., Smith, A. M., and Thomas, D. (2013). Galaxy Zoo 2: detailed morphological classifications for 304122 galaxies from the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society*.