Report for project 3

Jon Audun, Mikael Ravndal and Adam P W Sørensen

December 13, 2018

In this project we analysed some cooking data and did fairly well.

1 Introduction

In this project we explore a delicious classification problem: Given a list of ingredients, can you predict what cuisine the final dish will be? There are certainly cases where this is easy. If you meet you friend at the store and they have burritos, guacamole, salad, and beef in their basket, you can feel confident they are having Mexican food. But what if they have flour, yeast, and milk? They are probably baking, but are they making a French baguette our Indian naan bread?

In this project we will use various machine learning techniques to solve this classification problem. Our data comes from an old kaggle competition.

describe data here!!!

The paper is organized as follows: A section on methods, a section on selection and so on.

Machine learning has gained a huge popularity boost over the last couple of years. And this is no wonder: the techniques have a wide range of applications and can be a major asset if you know when to use what.

When to use what is exactly what we're going to have a brief peek into in this project. We will evaluate the performance of three different methods for classification on a data set consisting of recipies labeled with the cuisines they belongs to. That is we are looking at a classification problem with multiple classes. The methods we will look at are logistic regression, support vector machines and random forests.

All the code we have implemented can be found at `https://github.com/???`. A print of the jupyter notebook is also attached at the end of this report.

2 Methods

First we need to establish some notation. In the following we assume that we are given $N \in \mathbb{N}$ samples consisting of $p \in \mathbb{N}$ features and one target. Further we let $K \in \mathbb{N}$ be the number of classes in our classification problem. We then denote by $x_{i,j}$ the j-th feature of the i-th sample and by $t_i \in \{1, 2, \ldots, K\}$ the target of the i-th sample.

2.1 Reading in the data

The raw data lives in a .json file, which we load using Pandas. Each recipe has a cuisine and a list of ingredients. The first thing we do is to join all the ingredients into one string, and then we use scikit learns `CountVectorizer` on the corpus of all recipes to get a matrix representation of the data. The `CountVectorizer` works by making each individual word in the whole corpus a feature. Each recipe is then a vector in $\mathbb{R}^p$, with the k'th entry equal to the number of times the k'th feature (ingredient) appears in the recipe, usually this will be 0 or 1. This means that a recipe that calls for tomato sauce will have the features tomato and sauce set to 1, even if it does not use a tomato.

Before using the `CountVectorizer` we clean up the data a little bit, namely we replace hyphens with spaces, to make sure e.g. *low-fat* and *low fat* are treated the same, and we drop all numbers and special characters. The latter because the recipes are not all formatted in the same way, with some including amounts. When using `CountVectorizer` we set the parameter `min_df` to 3. This means we only include features that appear at least 3 times in all lists of ingredients. A feature that only appears once is useless for prediction, because it will either be contained only in the training data or only in the test data. We also excluded features that only appeared twice because the likelihood for them to appear in just the training data or just the test data are relatively high, approximately 68

2.2 Logistic Regression

Let $\boldsymbol{x}_i := (1, x_{i,1}, x_{i,2}, \ldots, x_{i,p})$ be the row-vector in $\mathbb{R}^{p+1}$ consisting of a one followed by the features of sample i. Also let $\boldsymbol{\beta_k}$ be a column-vector in $\mathbb{R}^{p+1}$ for $k \in \{1, 2, \ldots, K-1\}$ and let $\boldsymbol{\theta} := [\boldsymbol{\beta}_1 \ \boldsymbol{\beta}_2 \ \ldots \ \boldsymbol{\beta}_{K-1}]$ be the matrix with the $\boldsymbol{\beta}_k$'s as it's columns. In the case of multinomial logistic regression our model is then given by

$$\begin{aligned} P_k(\boldsymbol{x};\boldsymbol{\theta}) &= \frac{\exp(\boldsymbol{x}\boldsymbol{\beta}_k)}{1+\sum_{l=1}^{K-1}\exp(\boldsymbol{x}\boldsymbol{\beta}_l)} \qquad k \in \{1, 2, \ldots, , K-1\} \\ P_K(\boldsymbol{x};\boldsymbol{\theta}) &= \frac{1}{1+\sum_{l=1}^{K-1}\exp(\boldsymbol{x}\boldsymbol{\beta}_l)} \end{aligned} \tag{1}$$

where the $\boldsymbol{\beta_k}$'s are the coefficients we wish to estimate. $P_k(\boldsymbol{x};\boldsymbol{\theta})$ is then the probability that a sample with features $\boldsymbol{x}$ belongs to category k.

The way we fit our model in the case of logistic regression deviates slightly from the usual proceedure with the introduction of a loss function. We now instead define a function we wish to maximize, namely the log-likelihood function given by

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{N} \sum_{k=1}^{K} \chi_{\{k\}}(t_i) \log(P_k(\boldsymbol{x}_i; \boldsymbol{\theta})) \tag{2}$$

However the difference doesn't go further than the fact that this function shouldn't be interpreted exactly as a loss function. Other than that we proceed as usual by minimizing the negative of this function in order to maximize the original function. To do this we first need to compute the derivative of this function with respect to $\boldsymbol{\theta}$. Some computations gives us

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \beta_{k,j}} = \sum_{i=1}^{N} \chi_{\{k\}}(t_i) x_{i,j} (1 - P_k(x_i; \theta)) \tag{3}$$

We are now all set to use the gradient method of your choice to minimize $-L$ as a function of $\boldsymbol{\theta}$.

2.2.1 Gradient descent

The perhaps simplest approach to minimizing a real valued function f iteratively, is to set some starting point $\boldsymbol{x}_0$, compute the gradient ∇f in $\boldsymbol{x}_0$, and move $\eta > 0$ in the opposite direction of this gradient in order to obtain $\boldsymbol{x}_1 = \boldsymbol{x}_0 - \eta \nabla f(\boldsymbol{x}_0)$. Then one iterates by setting $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \eta \nabla f(x_0)$ for $i = 1, 2, ...$ This is the framework of gradient descent. For small enough η this implies $f(\boldsymbol{x}_i) > f(\boldsymbol{x}_{i+1})$. The problem with this approach is to find out what η is small enough. If we choose η too small, it might take days for our algorithm to converge. If we on the contrary choose η to big, it might not converge at all.

2.2.2 Newton-Raphson

The Newton-Raphson method is a kind of gradient descent method, except we choose the learning rate in a more clever way. We thus start out by choosing a random starting point $\boldsymbol{x}_0$, just as in the case of gradient descent. The update step however is a bit different:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \frac{\partial^2 f(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_i^T} \frac{\partial f(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i} \tag{4}$$

The result is a method that converges way faster than normal gradient descent. The draw back is however the need to compute the Hessian of your objective function. For a thourough derivation of this method consult [**?**]

2.2.3 Stochastic gradient descent

To avoid overfitting when training neural networks, training on randomly picked batches of the training data for each iteration is a good regularization method. If we let b be our batch size, the first step of this method consists of dividing the training data into $N//b$ batches. One does this by picking from the N data-points without replacement. The second step is then to perform a gradient descent as described in section 2.1.1 with a new batch for each iteration.

2.3 Support Vector Machines

This section is based on [James et al., 2013, Chapter 9] and [Hastie et al., 2009, Chapter 12].

2.3.1 Linearly separable data

Support vector machines where originally introduced to be used for a two class classification, so we will discuss that case first. For ease of notation let the classes be $\{-1, 1\}$. Recall that if $\beta_0, \beta_1, \dots, \beta_p \in \mathbb{R}$, then we can define a hyperplane H in $\mathbb{R}^p$ by

$$H = \left\{ \begin{pmatrix} z_1 & z_2 & \cdots & z_p \end{pmatrix} \,\middle|\, \beta_0 + z_1\beta_1 + z_2\beta_2 + \cdots + z_p\beta_p = 0 \right\}.$$

We can always assume that

$$\sum_k \beta_k^2 = 1,$$

and from here on out we will do just that.

We say that a given data set is linearly separable, if there exists a hyperplane H in $\mathbb{R}^p$ such that all data points $x_i = \begin{pmatrix} x_{i1} & x_{i2} & \cdots & x_{ip} \end{pmatrix}$ with $t_i = 1$ satisfy

$$\beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p > 0, \tag{5}$$

and all data points $x_j = \begin{pmatrix} x_{j1} & x_{j2} & \cdots & x_{jp} \end{pmatrix}$ with $t_j = -1$ satisfy

$$\beta_0 + x_{j1}\beta_1 + x_{j2}\beta_2 + \cdots + x_{jp}\beta_p < 0. \tag{6}$$

Intuitively all points with $t_i = 1$ lie on one side of H and all points with $t_i = -1$ lie on the other side. This intuition is entirely correct if $p = 2$, if $p > 2$ then it is still correct, provided one has the right mental image of the sides of hyperplane. Note that equations (5) and (6) can be combined to the single equation

$$t_i(\beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p) > 0. \tag{7}$$

Given some new data point $z = \begin{pmatrix} z_1 & z_2 & \cdots & z_p \end{pmatrix}$, we then predict $t \in \{-1, 1\}$ such that

$$t(\beta_0 + z_1\beta_1 + z_2\beta_2 + \cdots + z_p\beta_p) > 0.$$

2.3.2 Data that is not linearly separable

If a given data set is linearly separable, then there will be infinitely many separating hyperplanes. From one point of view, any one of these will do to make predictions. However, it seems intuitively correct to pick a hyperplane that is in the middle of the two classes. This is achieved by choosing the so called maximal margin hyperplane. That is, we want to choose $\beta_0, \beta_1, \cdots, \beta_p$ such that we maximize $M > 0$ under the condition that

$$t_i(\beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p) \geq M,$$

for all data points x_i.

Of course, or data will not usually be linearly separable. To still get useful classification, we will accept hyperplanes that do not cleanly separate the points. This is achieved by introducing for each data point x_i a slack variable $\varepsilon_i \geq 0$. Then aim to find $\beta_0, \beta_1, \ldots, \beta_p$ that maximize $M > 0$ under the conditions

$$t_i(\beta_0 + x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots + x_{ip}\beta_p) \geq M(1 - \varepsilon_i),$$
$$\sum \varepsilon_i \leq C,$$

where $C \geq 0$ is a tuning parameter. We note that if $\varepsilon_i = 0$, then the point x_i is outside the margin of the hyperplane, if $0 < \varepsilon_i < 1$, then x_i is inside the margin but on the right side of the hyperplane, and if $1 < \varepsilon_i$, then x_i is on the wrong side of the hyperplane, i.e. it is misclassified. Hence C is an upper bound on the number of point we are allowed to misclassify when we choose H. However, a choice of $C < 1$ is still very valid, it just corresponds to choosing a very narrow margin, is we do not allow many points to be inside the margin.

2.3.3 Kernels

Actually solving the maximization problem of finding a hyperplane H and a margin M is done using Lagrange multipliers. We will not discuss how this is done in detail, but will just focus on two aspect of this approach (see equation (12.17) in [Hastie et al., 2009])

(a) To find H and M one does not actually need the data points x_i but rather all inner products $\langle x_i, x_j \rangle$.

(b) H and M will not depend on all the datapoints x_i, the points they do depend on are called the support vectors.

The first of these points leads to the kernel techniques for support vector machines. In a variety of classification problems it can be helpful to change the feature space. This is done by finding some mapping $\phi\colon \mathbb{R}^p \to \mathbb{R}^q$ and then classifying $\phi(x_i)$ instead of x_i. A typical example is to add polynomial features, if $p = 2$ say, we might want to consider not only $\begin{pmatrix} z_1 & z_2 \end{pmatrix}$ but instead $\begin{pmatrix} z_1 & z_2 & z_1^2 & z_1 z_2 & z_2^2 \end{pmatrix}$. Computing all these extra features can be a time and memory consuming task, but for support vector machines one does not need to do it. Since we only need inner products of datapoints, it suffices to find a kernel function K such that

$$K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle.$$

For instance using the kernel function

$$K(x_i, x_j) = \sum \left(1 + \sum_{k=1}^{p} x_{ik} x_{jk} \right)^d ,$$

corresponds, upto some constants, to adding polynomial features of degree d.

2.3.4 More than two classes

To use support vector machines in a situation where there are $K > 2$ classes, scikit learn by default uses a so called one-vs-rest approach. For each class c we use a support vector machine classifier with classes $\{-1, 1\}$ chosen so that class c is coded 1 and all other classes are coded -1. This gives K hyperplanes with parameters $\beta_0^c, \beta_1^c, \ldots, \beta_p^c$, $c = 1, 2, \ldots, K$. Given an unseen datapoint z we classify it as belonging to the class c for which $\beta_0^c + \beta_1^c z_1 + \cdots \beta_p^c z_p$ is the largest.

2.4 Decision trees

Wikipedias overview on decision trees:

> A decision tree is a flowchart-like structure in which each internal node represents a “test” on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

A decision tree typically uses all of the features as its features. The result of this is then that you get a classifier which predicts well on the training data, but falls short on testing data. This is because it fits its classifier perfectly to the data which it has seen. So a lone decision tree with all features will almost always be overfitted.

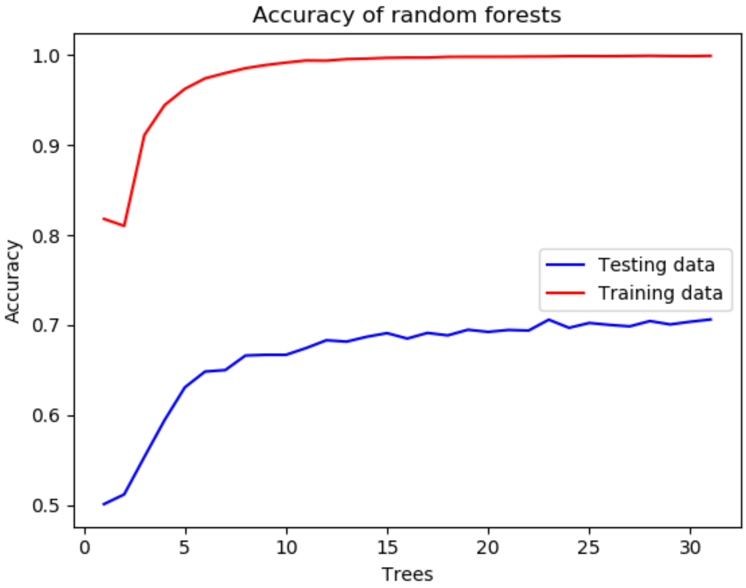
2.4.1 Rather a forest, then just a tree

This is why random forests are more popular. A random forest can usually generalise much better then what a lone tree can.
A random forest is constructed by choosing how many decision trees you want in the forest and then train every tree on your data. But the construction of these trees is done differently then a lone decision tree.

Each tree just use a subset of the features, typically $features = \sqrt{all - features}$. This makes it so each tree functions a bit differently and will then predict different things. The final prediction then becomes which of the classes which gets voted most for when every tree predicts their own.

2.4.2 Difference in number of trees

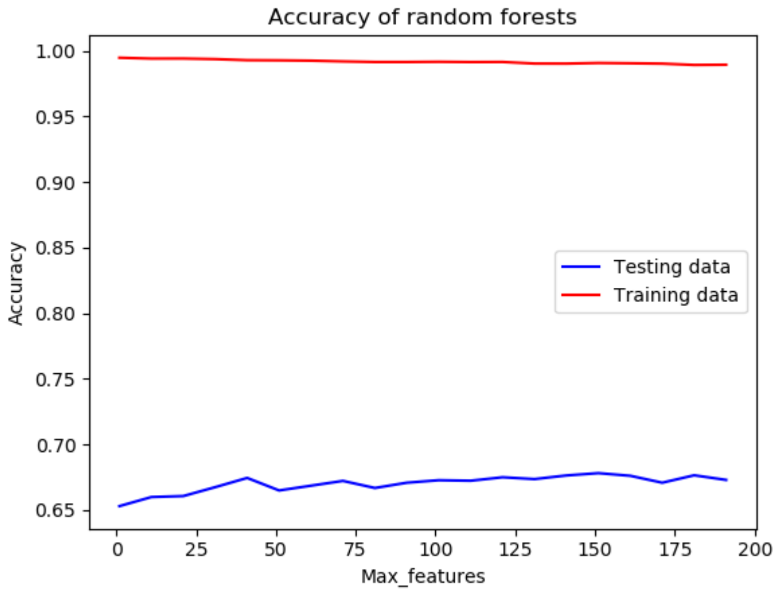
The number of trees in the forest is the main hyper parameter we have used to tweak this classifier. But it seems like the rule of thumb was the more, the merrier. Here is a plot of how well the random forest did:



So more trees provides a better accuracy. But the classifier starts to have a decent accuracy after 10 trees.

2.4.3 Difference in number of max_features

Here is a plot on how well the accuracy is using different numbers as max_features:



The number of trees used is 10.

The default in the scikitlearn classifier is using the root of the number of all the features as max_features. Since we have approximately 2000 features, the root becomes 44.72. That's why I computed different accuracies around this number.

The plot tells us that it doesn't seem to matter much what we use as max_features. The optimum in the plot is around 40 which gives us just more reason to stick with the default of using the root of all the features as max_features.

3 Model Selection and Verification

4 Results

5 Conclusion

We did good.

References

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning*. Springer Series in Statistics. Springer, New York, second edition.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Springer Texts in Statistics. Springer, New York.