Development of machine learning models at a customer level

Master of science internship

Max Halford

Supervisor: Adrian Foltyn
Referee: Jérôme Farinas
February 1st - August 1st 2017

Contents

Figures

Code

Acknowledgements

I would sincerely like to thank the whole staff of HelloFresh. The mix of nationalities and the good overall atmosphere made my stay in Berlin a memorable time.

I am grateful to the data science team and the head of my department for being patient with me and for having taught me so much. I am happy to have been considered somewhat of an equal to them and to have done valuable, lasting, and interesting work. I hope I have been worthy and that the things I have worked on will endure!

Chapter 1

Introduction

1.1 HelloFresh presentation

HelloFresh is a multinational meal delivery service with a presence in several Western European and North American countries, as well as Australia. The company was founded in 2011 in Berlin. HelloFresh sells meal-kits including ingredients and step-by-step recipes. The company delivers over 7.4 million meals a month to over 800,000 regular subscribers.[citation needed] Boxes are usually offered on a 3×2 (3 meals for 2 people) portion, with variations such as 3×4 (3 meals for 4 people) and 5×2 (5 meals for 2 people). The boxes are delivered weekly on a flexible subscription model.

From a business point of view, HelloFresh works on a subscription based model. Customers subscribe to HelloFresh and receive a food box each the week. The subscription is an opt-out one, meaning that customers explicitly have to tell HelloFresh that they do *not* wish to receive box any given week - by default they will. Although is it uncommon, customers may have multiple subscriptions. Customers may cancel their subscription at any given time.

1.2 Internship objectives

My internship revolved around building models that helped business to get a better understanding of the customers. A lot of the operational decisions that are made at HelloFresh are based on forecasts; the more we know about a customer's upcoming behaviour the more smoothly we can react to it.

First of all the main result I had to produce was a model that would forecast the churn rate for any given subscription. Secondly I had to put in place data science tools that would streamline my work and make it usable for other projects. Finally my last project concerned building a recommendation engine to improve customer experience.

Although HelloFresh is a relatively mature company – at least for a start-up – the data science team I joined was quite fresh and didn't have a lot of processes in place. Building a relationship of trust and open communication with the rest of the company was also a big part of my internship.

1.3 Data platform presentation

HelloFresh is a data-driven company where a lot of people need access to information under various forms. There are a lot of analysts that work for different departments and who need different pieces of information on a daily basis – especially those working in procurement and marketing. To respond to these needs a data engineering team was put in place quite some time before the data science team ever existed. All in all HelloFresh has a cutting-edge data platform that enables data scientists to access large amounts relatively easy.

HelloFresh's mature data platform relies on cutting-edge technology. Because HelloFresh is a young company the data engineering team relied on the latest technologies available.

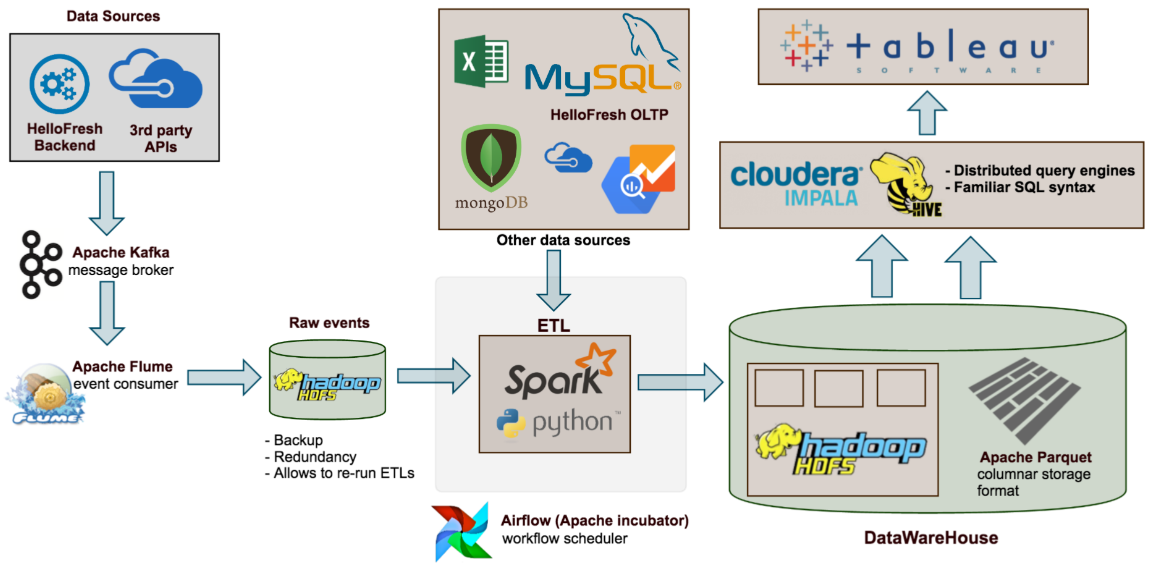


Figure 1.1: Data platform architecture

First and foremost the data produced by the website and the mobile app is transmitted to a Hadoop store through Apache Flume and Apache Kafka. This broker/consumer pattern enables fault-tolerance and can scale horizontally. The events produced by the app and the website are stored in a raw format that only the data engineers access. A custom ETL tool built on top of Apache Spark and Apache Airflow processes the raw data on a fixed schedule and makes it available in another Hadoop store that follows a snowflake schema – that is with fact tables and dimension tables. An Impala query engine is put on top of the final Hadoop store to provide an SQL-like querying language. Impala works in a similar way to Pig and avoids having to write map-reduce jobs in Java or Scala. The Hadoop store is connected to a Tableau server so that data analysts can easily produce visualisations and can re-sync them whenever the data is refreshed.

Chapter 2

Towards a single customer view

2.1 Introduction

At a high level, these are the goals HelloFresh has regarding customers:

- Prevent people from churning
- Encourage customer to spend more time activated than paused
- Make cancelled customers reactivate

To reach these goals we have to identify actionable metrics and determine what drives these metrics. The following questions are typically ones whose answers could be metrics that would have a real business impact.

“*How long until a particular non-cancelled subscription gets cancelled?*”

“*What is the probability for a subscription to be in each state in the upcoming weeks?*”

“*How long until a particular cancelled subscription gets reactivated?*”

The potential answers to these questions would be just as good as the data they are based on. To this day this single source of truth doesn’t exist at HelloFresh; the data is in many places and it is difficult to regroup without fearing the introduction of anomalies and inconsistencies. If we had a single source of truth aggregating data for each customer, then we could start building forecasting models, recommendation engines and whatnot. From a business perspective this single source of truth is called a *Single Customer View* (SCV), a holistic representation of the data a company knows about it’s customers. Implementing an SCV is as much a data engineering task as it is a data scientist one. Processes have to be put in place to transfer data from multiple sources into a single location, which is clearly a data engineering task. However, a data scientist approach has to be made to identify what and how data should be extracted for modeling purposes.

A big part of my work at HelloFresh involved building a tool that can be used to construct a SCV, the ensuing tool is called **Cerebro**. I then used Cerebro as part of a churn model I describe in the next chapter. Moreover my goal was to build a tool that could be used and maintained by other data scientists.

2.2 Conceptual ideas

An SCV is usually obtained by aggregating data from different sources. Formally, for each subscription $i \in \{1, \dots, n\}$ we associate a set of features f_{ik} with $k \in \{1, \dots, p\}$, thus obtaining a table of size $n \times p$ with the following shape.

$subscription$	$feature_1$	$\dots$	$feature_k$	$\dots$	$feature_p$
1	$f_{1,1}$	$\dots$	$f_{1,k}$	$\dots$	$f_{1,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
i	$f_{i,1}$	$\dots$	$f_{i,k}$	$\dots$	$f_{i,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
n	$f_{n,1}$	$\dots$	$f_{n,k}$	$\dots$	$f_{n,p}$

The definition of an SCV is not a hard one to understand; the difficulty attached to an SCV lies in it's construction and it's accuracy. Although some features such as a customer's age can be obtained without much effort, other features can prove more difficult to construct. Moreover, although it may seem obvious, it is critical that the SCV contains the less discrepancies with the original data as possible. However this is true for any ETL process - to which the construction of SCV belongs. Because the SCV will then be used in data science pipelines each feature is supposed to be transformed in a way that it can be provided to a machine learning model. Indeed, in practice a data scientist spends a lot of time devising functions which are often the same. The preprocessing that has to be applied to certain features is often the same from one project to another, thus incorporating it in the SCV can save a lot of time that can be spent on higher business value tasks.

To do forecasting we would need to use a past SCV, train a model on it and then apply it to a current SCV. This requirement adds another dimension to our SCV representation: the SCV has to be time *time aware*. A simple way to do so is to add a column to the previous table to indicate the date as is done in the following table. It has to be said that we not considering streaming applications but rather focusing on batch ones – because this is HelloFresh's use case. Every customer has a feature that changes over time – the most simple example being their age – but we reasonably consider that each feature only changes on a daily basis.

$subscription$	$date$	$feature_1$	$\dots$	$feature_k$	$\dots$	$feature_p$
1	1	$f_{1,1,1}$	$\dots$	$f_{1,1,k}$	$\dots$	$f_{1,1,p}$
$\dots$	1	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
i	1	$f_{i,1,1}$	$\dots$	$f_{i,1,k}$	$\dots$	$f_{i,1,p}$
$\dots$	1	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
n	1	$f_{n,1,1}$	$\dots$	$f_{n,1,k}$	$\dots$	$f_{n,1,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
1	j	$f_{1,j,1}$	$\dots$	$f_{1,j,k}$	$\dots$	$f_{1,j,p}$
$\dots$	j	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
i	j	$f_{i,j,1}$	$\dots$	$f_{i,j,k}$	$\dots$	$f_{i,j,p}$
$\dots$	j	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
n	j	$f_{n,j,1}$	$\dots$	$f_{n,j,k}$	$\dots$	$f_{n,j,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
1	d	$f_{1,d,1}$	$\dots$	$f_{1,d,j}$	$\dots$	$f_{1,d,p}$
$\dots$	d	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
i	d	$f_{i,d,1}$	$\dots$	$f_{i,d,j}$	$\dots$	$f_{i,d,p}$
$\dots$	d	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
n	d	$f_{n,d,1}$	$\dots$	$f_{n,d,j}$	$\dots$	$f_{n,d,p}$

To resume we would like to put in place a database to accommodate p features for n subscriptions at d dates. $f_{i,j,k}$ represents feature k for customer i at date j. In production the last data d will represent the latest information we have on each customer. To produce predictions we would apply a machine learning model on the features associated to d. The machine learning model would be trained on dates $\{1, \dots, d-1\}$. Finally at HelloFresh we also add a column to distinguish the country each customer belongs to for practical reasons.

The SCV is the missing link between the data warehouse and the matrix that is fed to a machine learning algorithm. It provides an abstraction that makes ensuing data science pipelines much simpler. Indeed, mixing the data gathering and processing with the actual machine learning part of a project can lead to spaghetti and coupled code which inevitably becomes a nightmare to maintain. Data scientists do not always a deep knowledge of design patterns and typically more emphasis on “getting the job done” instead of writing “good” code. Separating the data gathering from the rest of the pipeline helps in reasoning about a data science project in terms of components which each have a specific responsibility. The end goal at HelloFresh is that all the data gathering required for customer-level machine learning models is done through Cerebro.

Finally the obvious goal of Cerebro is to respect the DRY principal – Don't Repeat Yourself. The data science team at HelloFresh works on a lot of models and there is always some overlap as to what kind of data we want to look at. The idea with Cerebro is that once the data has been collected it can be reused by different models without having to rewrite a query. This avoids a lot of mistakes and saves a lot of time. In other words Cerebro acts as the single source of truth the data scientists should be using for their customer-level models.

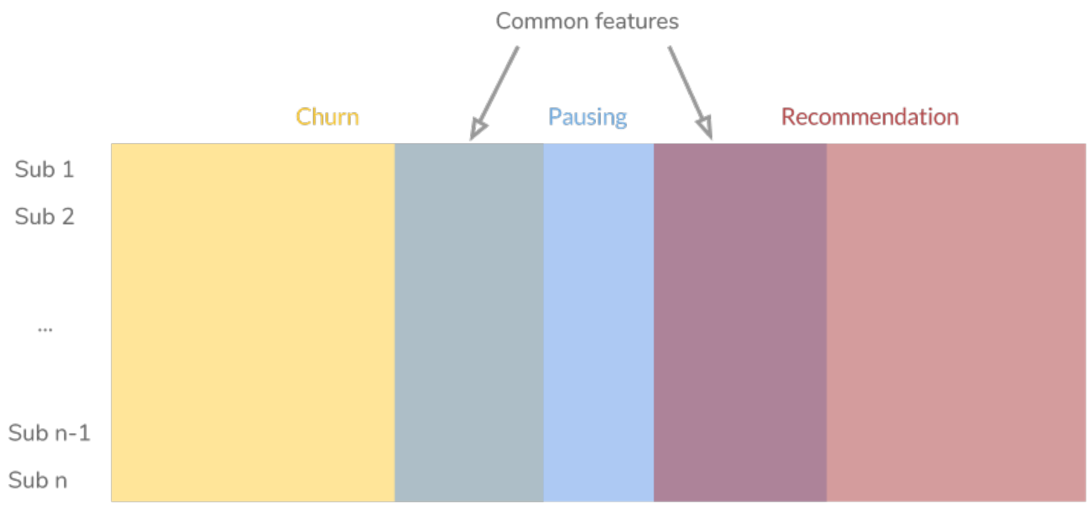


Figure 2.1: Different models may use the same features

Uber seems to do something similar for their machine learning [4]. However Uber has a larger team and an extremely strong data engineering department. Conceptually, the main difference is that Uber also has real-time needs that they address in their solution. In practice they have two "feature stores" as they call them, one for real-time and one for batch processing. For the real-time part they use Cassandra and for the batch they use Hadoop with Hive. They use the Cassandra for fast retrieval for their real-time models and they regularly migrate the data from Cassandra to Hadoop. All in all it's a paradigm that does not have silver bullet solution to it yet, but patterns are emerging.

2.3 A single unit of work: the fetcher

There is no dependence between each column in the single customer view. Each column represents a feature that should be able to obtained independently from the rest of the columns. This property which is simple and somewhat natural leads to efficient parallelism of code and execution, which is core to how Cerebro is built.

In Cerebro a *fetcher* is responsible for producing one or more features of the SCV. The fetcher contains code for querying a data source and preprocessing the result of the query into a form that is usable by a machine learning algorithm.

Because all the fetchers need is access to a data source they are independent from each other. This way the fetchers can be run concurrently whilst preserving idempotency. Moreover, each fetcher's source code can be written in a separate file which makes Cerebro extremely more easy to maintain and understand.

In practice – code-wise that is – a fetcher has to provide a `fetch` method which can take temporal and geographical arguments into a account. For example in the following Python snippet the `fetch` method has to return results that exist at a given date for a given country. To do so the fetcher can access the `source` it has been provided with and run a query that includes predicates which filter according to the `date` and `country` arguments.

```python
class Fetcher():

    name = None

    def __init__(self, source=None):
        self.source = source

    def fetch(self, date, country):
        raise NotImplementedError
```

Listing 1: Fetcher interface

Because Cerebro is implemented in Python the `fetcher` method is supposed to return a `DataFrame` from the `pandas` library which is one of the *de facto* data processing libraries in Python. However Python is not a statically typed language so it's possible to enforce the return type, but it's fine to impose this as a soft constraint that be checked during a code review. Cerebro is meant to be a tool for assisting a data scientist, not a whip that imposes a methodology.

The reason why the `fetcher` method should provide a dataframe and not a series – pandas also has a `Series` class – is because some fetcher might return multiple features, for example when obtaining the address of a customer where it would be inefficient to extract each feature separately.

Finally, the dataframe returned by `fetcher` method should be indexed by the subscription ID, the date and the country as was described in previous section. Intuitively this will permit to join the dataframes produced by the fetcher to obtain the SCV.

2.4 Storage

Once each fetcher runs it should store it's output in a database which can then be accessed by a data science application. Because the SCV is, conceptually, "only" a table we can do this in many ways.

Although the SCV has until now been represented with a tabular form, it's important to understand it is a *concept* and does not necessarily be stored as a single table. For many reasons a table and specifically a relational one doesn't suit our needs. First off a RDBMS expects there to be a predefined schema which is too strong a constraint. For example we want to want to add a feature and an associated fetcher we would have to edit the table and add columns. Indeed as time goes on we want to be able to add – and maybe remove – fetchers without much difficulty. First of all having to manually maintain the existing columns and the new ones to be added adds more complexity than is needed and removes the transparent feeling of how the SCV should be stored.

The fact is that the columns change over time because new information becomes available or because some other information becomes unavailable, this is what happens in modern real-time applications and is a headache many data scientists and engineers have to deal with. The relational model is simply too clunky and too structured to deal with this fact in a smooth manner.

After some research I settled on using the Hierarchical Data Format system [3], which is commonly denoted by "HDF" followed by a version number – the most recent release is HDF5 – and should not be confused with the "HDFS" acronym which stands "Hadoop Distributed File System" and is also a file system. The HDF system has historically been used by academics for handling large amounts of data. Without going into too much details it provides an abstraction for storing and querying numerical data efficiently. Although it somewhat breaks the program/data independence principle and doesn't provide a declarative syntax such as SQL, it largely suffices for numerical applications.

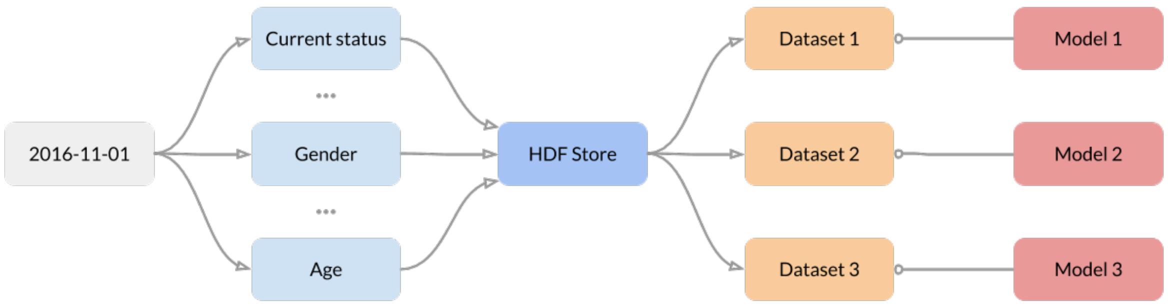


Figure 2.2: Cerebro data flow

At it's core the HDF system stores data in a `*.h5` file. It can store arbitrary data in a tree-like manner, much like the way files are stored in UNIX systems. Typically each leaf of the storage tree is a dataframe which is indexed with a b-tree to provide efficient querying. Any application storing data in an HDF file has to provide a path where to store the data. In our case the path can be composed of the country, the date and the name of the fetcher that is producing results. For example the `US/2016-04-20/customer_age` would hold a dataframe containing the age of each customer in the US that existed the 20^{th} of April 2016. By running multiple fetchers at a given data we can store them all under the `US/2016-04-20/` path. The ensuing data science pipeline can then retrieve the data it needs separately.

Effectively the HDF "store" as it is usually acts as an intermediate storage. The advantage of having this "link" between the data gathering and the rest of the data science pipeline makes it possible to run both processes concurrently with the HDF store being available on a shared location.

It's important to understand that each fetcher provides a *template* for accessing data and runs given some *context* – e.g. a date and a country. In the diagram above a set of fetchers can be run for the given data "2016-11-01" and will store their results in the HDF store according to the data and their name. Of course this is also true if a date is provided, the graph omits it for simplicity.

Once the data is stored in the HDF store it can be retrieved via client libraries – the recent versions of Python's `pandas` have a predefined method to do so – by joining the dataframes produced by each fetcher. Having the data naturally by this path system makes retrieving data for a particular segment – i.e. a date and/or a country – very easy and extremely scalable. Indeed each dataframe that is stored only contains as many rows as their are subscriptions and at most a handful of columns which can consequently easily be manipulated in memory. It's fairly to obtain historical data for training purposes by iterating over a range of dates.

2.5 Sources

One of Cerebro's features is to act as a *mediator* to aggregate data from heterogeneous sources. In the end all that matters is that the data is stored in a tabular format. HelloFresh has a microservices architecture that uses multiple storage options to function. Since a few months a lot of work has been put into transferring the meaningful data in a Hadoop data warehouse, however a lot of useful information is still only located in application databases running on MySQL, PostgreSQL, MongoDB and what not. To solve this problem *adapters* are made available to each fetcher so that they can query any source and obtain a `pandas` `DataFrame` instance.

During my internship I interacted with the following data sources:

- **Application relational databases**: Most of the customer related information is stored in Amazon RDS instances that run PostgreSQL or MySQL. Accessing these is trivial and only requires writing an SQL query which can be run through `pandas`'s `read_sql` method.
- **Hadoop Impala data warehouse**: HelloFresh runs a Hadoop data warehouse which can be accessed via the Impala query engine which provides an SQL-like language for retrieving data – which is much easier to write than map-reduce jobs! – therefore it can also be used via `pandas`'s `read_sql` method.
- **Google Big Query**: Towards the end of my internship I began extracting features from Google Analytics data which is made available through Google Big Query for premium accounts. Luckily during my internship a new version of `pandas` was released which provided a `read_gbq` method for running queries against a Big Query table.

2.6 Implementation

2.6.1 Architecture

Cerebro is implemented in Python and stands on the should of a few popular libraries, namely:

- `click` for providing a command line interface which will be described in one of the following subsections.
- `pandas` for manipulating dataframes.
- `SQLAlchemy` for interacting with the MySQL, PostgreSQL and Impala query engines.

I did a lot of work to make Cerebro a scalable tool, both in terms of the volumes it can process and the speed at which new features can be added. As mentioned the fetchers are all independent from each other so that we run them in parallel, and this is easily doable with Cerebro.

Basically each fetcher has an associated Python file. For example the logic for obtaining a customer's age is written down in the `customer_age.py` script. All these scripts are kept in a folder aptly named `fetching`. The scripts have no other logic than the data gathering and preprocessing they are expected to do. It is the responsibility of an external application, such as a command line interface or a REST API, to run them in any desired way.

A bunch of utility methods exists in a folder named `util` that is kept separate. The most noteworthy function I implemented would run an SQL query against a list of MySQL instances and aggregate the results in a single dataframe. HelloFresh has multiple databases for many countries and it is extremely beneficial to automate this kind of work.

2.6.2 Fetcher example

Usually fetchers< do not exceed 150 lines of code – including whitespace and comments. For privacy reasons I'm only allowed to show a generic fetcher but it gives a good example of what the rest of the ones I implemented look like – I implemented 39 of them. The following snippet contains the code for the fetcher named `customer_age`; the name is quite explicit and the fetcher does nothing more than output a dataframe which contains the age of each customer. The subtlety is that the age is relative to the date the fetcher is run at, and this variable is outside of the fetcher's scope. Moreover, the age that it output can directly directly be plugged into a machine learning algorithm without requiring anymore processing – this would be different if the date of birth was returned. Although a very simple example, the `customer_age` shows some of the core concepts of Cerebro and serves the quite well when explaining Cerebro to fellow data scientists.

```python
import datetime as dt
from dateutil.relativedelta import relativedelta

from .base import Fetcher
from .util import run_dwh_query

class CustomerAgeFetcher(Fetcher):

    name = 'customer_age'

    def fetch(self, country, at, since=None, until=None):
        query = '''
        SELECT
            sd.subscription_id,
            MIN(cd.birthdate) AS birthdate
        FROM
            dimensions.subscription_dimension sd
            LEFT JOIN dimensions.customer_dimension cd ON
                sd.customer_id = cd.customer_id AND
                sd.country = cd.country
        WHERE
            sd.country = '{country}' AND
            sd.fk_created_at_date <= {at_date}
        GROUP BY
            sd.subscription_id
        '''.format(
            country=country,
            at_date=at.strftime('%Y%m%d')
        )

        df = run_dwh_query(self.dwh_conn, query)

        def get_age_in_years(birthday):
            try:
                datetime = dt.datetime.strptime(birthday, '%Y-%m-%d')
                 return relativedelta(at, datetime).years
            except (TypeError, ValueError):
                return None

        df['customer_age'] = df['birthdate'].map(get_age_in_years)
        df.drop('birthdate', axis='columns', inplace=True)

        return df
```

Listing 2: Customer age fetcher

2.6.3 Development scalability

Data scientists do not necessarily have a strong background in software engineering, in fact this seems to be quite rare in my experience. In a perfect setup the data engineers are supposed to take care of putting data pipelines in production. However, HelloFresh's particular setup and growth rate requires data scientists to handle some of the production aspect themselves. Moreover, Cerebro is effectively extracting features that are thought by the data scientists, hence it makes a lot of sense that it be the data scientists who deploy it and maintain it.

The way Cerebro is thought out means that each fetcher can be kept in a separate file. This makes easy to debug a fetcher in case the features it produces seem corrupt. In practice it only a few lines of code to run a fetcher inside a Jupyter notebook where it is quite convenient to debug tabular results.

2.6.4 Command line interface

Cerebro can be used with a command-line interface powered by the Python `click` package. Having a command-line interface available makes it easy to schedule cron jobs. The main command of the CLI is the `fetch` method which grabs data for given countries and dates.

Example usage

The most simple use case is to run all available fetchers for a given country and date.

```
python cli.py fetch US -d 2016-11-01
```

You can run the fetchers for multiple countries and dates.

```
python cli.py fetch AT DE -d 2016-11-01 -d 2016-11-02
```

You can run fetchers in parallel using threads.

```
python cli.py fetch AT DE -d 2016-11-01 -w 4
```

You can choose which fetchers to run.

```
python cli.py fetch US -d 2016-11-01 -c gender -c customer_age
```

You can also ignore certain fetchers.

```
python cli.py fetch GB -d 2016-11-01 -i current_status
```

CLI behavior

By default fetchers will not be run if they have already been run at the provided date(s) and country(ies). Before running the fetchers Cerebro filters out the ones that have already been run. The `-o` flag can be used to force data fetching. Implementation-wise Cerebro looks at all the fetchers that exist, then it removes the fetchers that were ignored or chosen in the command-line arguments, finally it removes the ones that exist by looking at the existing paths in the HDF store.

The fact that the command-line tool does not re-run the fetchers at the given date and country if the data has already been gathered proves extremely useful for handling mishaps. Indeed because the data we are collecting comes from a development database it's possible for it to be in maintenance or for one or more tables to be unavailable, rendering querying impossible. Running 40 fetchers without any errors every day under these conditions is unlikely and has to be accounted for. To handle these mishaps I initially put in place a small SQLite database to record what jobs failed to run. I then created a `backfill` command which would simply loop through the "missing" jobs and run much like they were run with the `fetch` command – effectively some code was reused for consistency. However, maintaining and syncing this backfill database with the state of the HDF store proved a lot of work for a job that was supposed to be a fix. In the end I settled on having multiple cronjobs running sequentially throughout the day to give all the fetchers a good chance of running. Although this way of doing leaves some room for error it suited our needs more than adequatly. Up until I left all the fetchers that were run on Tuesdays ran successfully after three cronjob runs, with 90% of them running with the first cronjob run. For example the following crontab file would run all the fetchers for Great Britain on Tuesdays by fetching the data of the previous Monday; the crontab would run at 2AM, 6AM, 10AM and 2PM server time to ensure all the fetchers were run at the end of the four runs.

```
MAILTO=data-scientists-global@hellofresh.com

CEREBRO_PATH=/home/scientist/data-science-cerebro
PYTHON=/home/scientist/anaconda3/bin/python

# Churn feature extraction
0 2 * * TUE cd $CEREBRO_PATH && $PYTHON cli.py fetch GB -w 4 -d
↪  $(date --date yesterday "+\%Y-\%m-\%d")
0 6 * * TUE cd $CEREBRO_PATH && $PYTHON cli.py fetch GB -w 4 -d
↪  $(date --date yesterday "+\%Y-\%m-\%d")
0 10 * * TUE cd $CEREBRO_PATH && $PYTHON cli.py fetch GB -w 4 -d
↪  $(date --date yesterday "+\%Y-\%m-\%d")
0 14 * * TUE cd $CEREBRO_PATH && $PYTHON cli.py fetch GB -w 4 -d
↪  $(date --date yesterday "+\%Y-\%m-\%d")
```

Listing 3: Example crontab file for running Cerebro

2.6.5 Optimising for speed

From the start Cerebro was thought out for being a scalable tool. In practice the data scientist team would like to run it one day to collect data from the previous day in order to update a model. This 24 hard-limit has to hold no matter the number of fetchers that have to be run.

Essentially by running each fetcher in a separate process all the fetchers can be run much faster than by running them sequentially. However, 40 core servers are not easy to come across and even so they are too expensive or have to be setup manually. Thankfully modern CPUs allow using a lot a large number of threads, 40 being a rather low number.

The two main bottlenecks for Cerebro are not computing power nor the number of threads a server can run. First of all the number of Hadoop workers is capped so as not to overload the servers the HDFS files and the map-reduce jobs are located on. Indeed the whole of the queries at HelloFresh are run by the same pool or workers which can run a maximum of around 50 jobs concurrently. Secondly while the queries are running they transferred back to the client in a producer/consumer pattern, hence running a lot of fetchers in parallel can introduce a network bandwidth bottleneck.

The number of threads that are used by Cerebro can be controlled in the CLI via the `-w` parameters – the `w` standing for *workers* – as can be seen in the previous crontab snippet. In practice I found that running four threads made it possible to run 40 fetcher in just over four hours, which was more than sufficient for training a machine learning model the rest of the day. Of course these numbers were true for Great Britain, I expect them to differ a lot for the United States where the data is much larger.

2.7 Improvements

I'm very happy to have been able to work on Cerebro. While I used it to collect features related to churn, the data science team at HelloFresh is beginning to use it for freebie fraud and customer-level forecasting. I was quite surprised to see that companies like Uber were using the same pattern for their machine learning models. In the future I would like to continue working on this pattern and include streaming data. Moreover a lot of Apache incubating projects seem to solve some of these issues and it might be worth looking into them. Finally I firmly believe that having some sort of dashboard with clickable commands would help popularise Cerebro internally. Generally I find that tools like Cerebro should be encouraged and developed as much as pure machine learning libraries are, indeed the fact is that in practice data scientists spend most of their time manipulating data, not training models, hence they should be empowered with the right tools.

Chapter 3

Predicting customer churn

3.1 Business case

Like many companies [1] – and especially in subscription-based ones – *churn* is used an indicator of how well a company is doing. Because of all the marketing and acquisition costs a customer at HelloFresh only becomes profitable after having payed for more than 3 boxes. Ideally we would like to know as fast as possible when a customer is going to terminate his subscription. Moreover we would like to know this as much as possible before the customer actually cancels in order to send him a prevention email – possibly with a discount voucher.

From my understanding there are two types of models a data science team can provide business teams:

- Static models that explain an output based on inputs
- Predictive models that can be used to take preventive actions

In our case the CRM (Customer Relation Management) team explicitly demanded from us a predictive model whose output they could plug into CrossEngage, the cross-channel marketing platform they use for email campaigns. Of course, a predictive model can also give some information on what drives the output – that is the most important features. Regardless, the point was that using complex models was not as issue as long as the performance of the said models was good. In other words I was tied to having to use linear models for the sake of comprehension.

The CRM team, and the other marketing teams in general, specifically wanted to know the probability of a customer to cancel in the next few weeks – the definition of churn is rather simple for subscription-based companies! Naturally whatever the methodology this task is easier for customers who have been subscribed quite some time, mostly because they have spent more time using – or not using! – the website and the application.

For the proof of concept we settled on determining the probability for a customer **who has received more than three boxes** to **churn anywhere in the next**

two weeks. Although this may seem quite restrictive, it still allows training on a rather large amount of data.

I split my work on this task in three parts. First I spent three weeks – this was the start of the internship so I had a lot to learn – exploring the data and making a list of what features could potentially be useful. Specifically I wrote everything down in a spreadsheet that stakeholders could see in order to follow my progress. Next I spent most of my time extracting the features in order of subjective importance thanks to Cerebro. Finally I built the machine learning model and the deployment code in parallel – I updated the model as I added features so as to see how well I was doing.

3.2 Training calendar

The fact that the model being built is predictive and has to be updated on a weekly basis implies a temporal restriction that to be kept in mind when determining what features to use. First of all, only features that can be computed at the most recent point in time can be used. Indeed, the goal of the model is for it to be applied to the latest data in order to obtain valuable predictions. Although some features might be available during training on past data, they might not be available right away for the latest point in time.

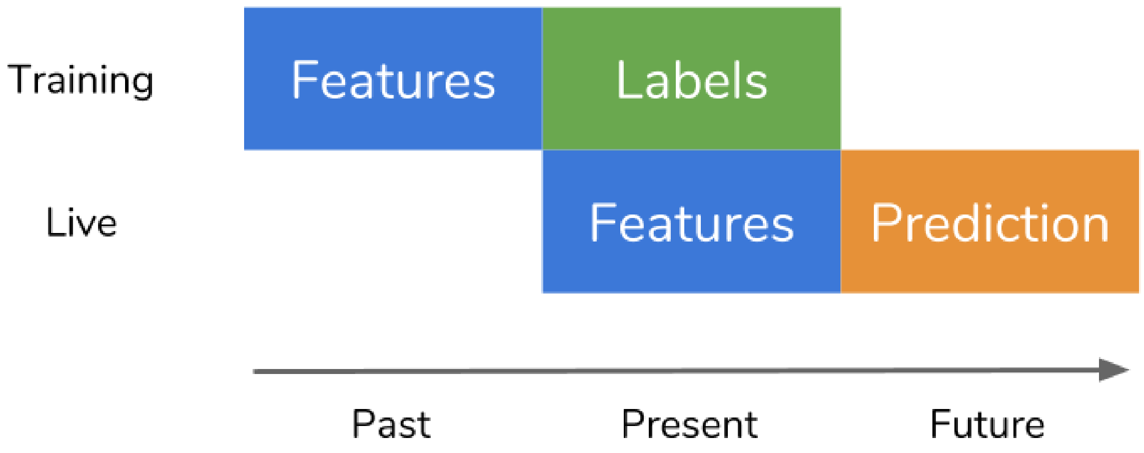


Figure 3.1: A typical learning timeline

First of all some features might simply be *leaking*, that is they may be carrying information that isn't supposed to be known at a certain point in time. For example, it's very to find features from the past that can explain why something happened, however this retrospective approach can't be used to predict the future. Data leaking can be quite subtle and it takes some discipline so as not to introduce it in a dataset. It is the bane of many Kaggle competition participants who fail to understand why their local cross-validation scores are significantly lower then their public leaderboard score. In our case it is very difficult to know what features will be available in the present, it takes a lot of domain-knowledge and questioning to make sure of this.

Second of all their might be some features that *should* be available at a certain point in time but are not because of technical reasons. For example at HelloFresh most, if not all, of all website and application events are transmitted to a Kafka or RabbitMQ queue before being stored permanently in a database. In other words, an event that occurs isn't necessarily recorded in real time. Again this is difficult to know and in practice I simply hoped for the best and crossed my fingers when I ran my model in production – and it worked!

The previous diagram displayed a typical timeline used when putting a predictive model in production. Effectively the model is trained on labeled instances that belong to the past. Afterwards the model can be applied to current instances in order to predict future labels. Of course this is a general concept that can applied to both classification and regression tasks. On a side-note, the blue blocks correspond to slices of the SCV at a certain point in time. However, at HelloFresh there is the concept of *cut-off* dates that adds some complexity to this timeline.

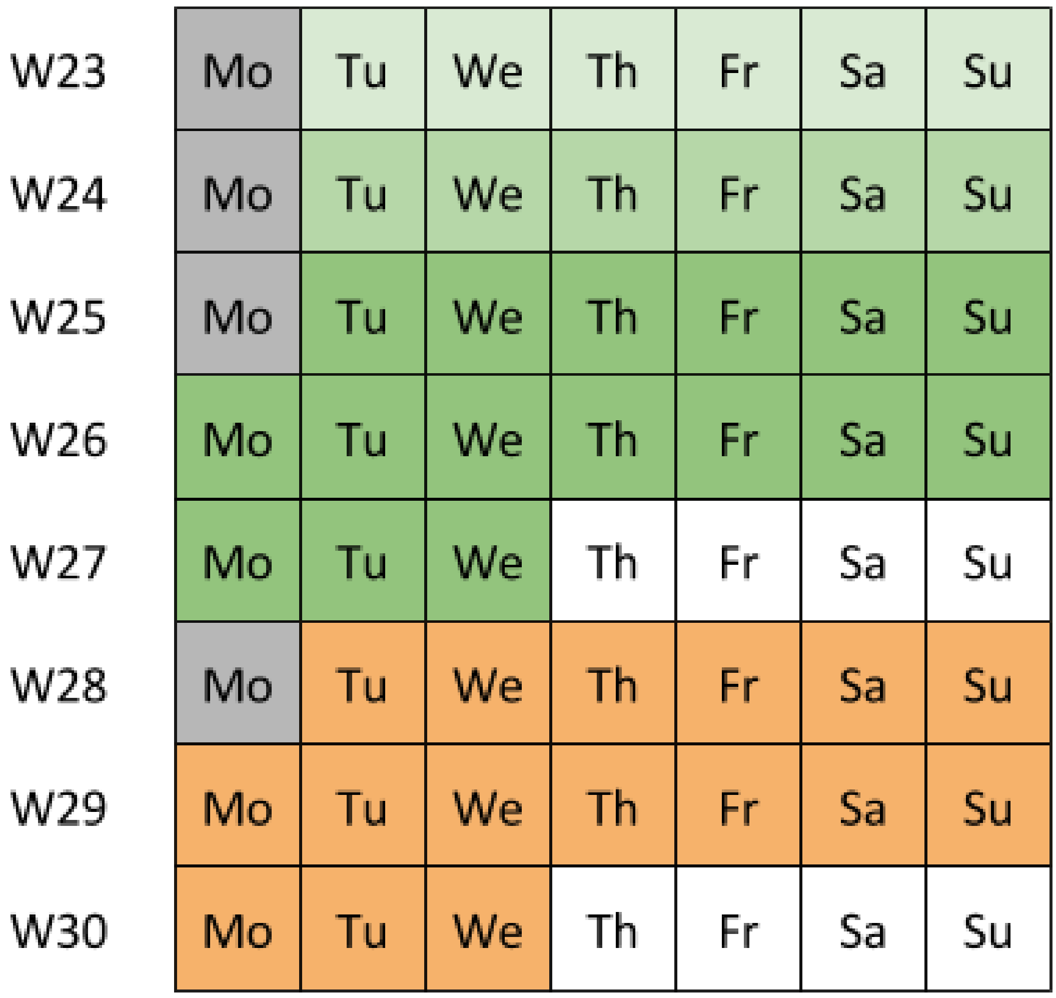


Figure 3.2: Learning timeline calendar taking into account cut-off dates

Without going into too much detail customer's may only pause or cancel for the upcoming week on the Saturday preceding the upcoming Wednesday. For example, if a customer is on a Friday, then he can pause or cancel for the upcoming week. If it's a Saturday, then he can't cancel for the upcoming week and has to receive a box. In other words, the status for the upcoming week is "blocked" between Saturday and Wednesday – inclusive – whereupon the new week starts. What this means is that the definition of two weeks CRM gave to the data science adapted has to be adapted to "wrap around" the two next weeks the status can be changed. This can be seen on the previous diagram. The idea is that we want to predict the probability of churning on Mondays, which corresponds to the grey squares. Obviously the Monday data is only accessible on Tuesdays, which is when the data can be collected with Cerebro – or some other medium, it doesn't matter. Because the status for the current week can't be changed yet, effectively the next Wednesday status is static, the upcoming two weeks time span *includes* the Wednesdays and stops on the Wednesday in two weeks, not the Tuesday! This is a bit complicated to understand and it is due to the specific nature of HelloFresh's business.

Nevertheless, because of this specific timeline issue, the training period has to be set one week further back in time. Indeed, by looking at the previous diagram it should be quite apparent that if the latest training Monday was the one of week 26 then the two week horizon would overlap with the current future. Luckily this three weeks step doesn't apply for the other weeks in the training period, indeed their respective two weeks horizon overlap with each – this is easier to see by observing the previous diagram.

Naturally we expect closer weeks to be more representative of the current period, simply because the website and the app will be more similar to the current version, which is extremely important. For the sake of simplicity – but also for practical reasons – we settled on considering the past six weeks as the training period. The trick is that identical customer's will overlap because if they were active one week and didn't churn then they will be part of the following week's training set. Theoretically one single customer could be represented six times in the training set. However this doesn't mean that there will be duplicates in the training set, simply because of the fact that some features are dynamic in nature, e.g. the number of received boxes or the total time spent on the website.

Getting our head around the timeline proved to be quite difficult. However it was a really interesting problem to deal because it is a dilemma that frequently occurs in live machine models. It's very difficult to find literature or online articles on how companies deal with these problems at scale. More generally the concrete implementation and deployment of live predictive models does not seem to be a popular subject. Of course it is not worthy of being researched by scientists but some framework would be more than welcome. I will get back to this in my conclusion but this is where I learnt the most during my internship. Carefully manipulating temporal data so as to introduce data leakage and making sure the model will work in production proved to be strong skills to possess. Moreover, these temporal issues are extremely hard to debug and do not necessarily ring any bells, they simply will reduce live performance and befound the modelers.

3.3 Making use of the single customer view

The goal is to predict churn for each customer/subscription – this is close to being a one-to-one relationship at HelloFresh. All the features we want to collect belong to a single subscription and thus fall in the net of the single customer view.

Once I determined which features would be of potential interest I started developing Cerebro before coming back the implementing the churn model itself. Without going much detail here were the broad families of features I collected.

- *Personal information*: for example the age and gender of person. This is an opt-in question at HelloFresh so it isn't always available, nevertheless the use of a boolean value indicating if yes or no the customer gave the information proved useful.
- *App interaction*: capturing the time spent on the mobile application. This proved to be useful in many: a decrease in average time spent on the app week on week could predict churn quite well. Alas I didn't the time into much depth but my gut feeling is that there is a lot that can determined by a customer's lack of interaction with the app.
- *Box information*: this is specific to HelloFresh but I simply used the size and type of the box a customer is receiving as features. My intuition was that customers with bigger boxes might have a bigger need for it. Then again cheaper boxes might encourage customers to stay.
- *NPS information*: **N**et **P**romoter **S**cores are extremely important in every business. They are often regarded as strongly representative KPIs as to how the business is doing in general. It is common knowledge at HelloFresh that NPS is correlated with churn.
- *Email interaction*: HelloFresh's CRM team can be quite aggressive in the way they send emails to people. Having some metric to track if a customer is being "spammed" or not can be useful. Moreover a nice feature I could extract was if a newsletter email – which works with an opt-out mechanism – was put in the spam box.
- *Errors*: during a box's lifetime a lot of mistakes may occur. Mostly the box can be missing one or more ingredients, we call these "Pick and pack" errors. Errors can also occur during the so-called "last mile delivery". If a customer spots a mistake, he can either call customer care and report the error, which is then tracked and accessible through the data platform. However, if the customer doesn't report the error then we are dealing with a false negative information which can be detrimental to a learning algorithm – i.e. there won't seem to be any link between churn and errors although in fact there is.
- *Acquisition detail*: what is common for many customer-facing companies stands true for HelloFresh: vouchers with price reductions tend to attract short-stayers who in our case leave after just one box. There is a rather large negative correlation between the voucher amount and the amount of time a customer stays

with HelloFresh. Moreover the acquisition channel seemed to be quite helpful in predicting churn, for example customers who joined after seeing a TV add tended to leave quite early – we can track "where" the customers come from by adding some leading characters to each channel's voucher codes, this is common practice in multi-channel marketing.

I have to say that it was difficult to get started on this work. This is completely the opposite of a Kaggle competition setting where the features are given to the data scientist. Here all what is available is a set of databases and a bit of domain knowledge, that's it. Although extracting the features seems more like a data engineering task, it also takes a data scientist to determine if the collected features carry mistakes in them or not. Albeit being grudging work, having Cerebro to work with helped a lot.

Sadly I didn't get to work on the second type of features described in [1]. They construct a set of features by applying vec2vec type algorithms to a large amount of click and interaction information. Although these kind of methods can be a bit of a black box, being able to extract meaningful features through embedded with raw features can save a lot of time. Regardless the data I collected was sufficient to get started.

As said all this work was developed with Cerebro. The really nice feature is that all the work I did will not have to be redone by a new joiner because all the data fetching is done in one single place! All in all I wrote around 40 fetchers which represented around 220 features. The reason why this number of features might seem high is that some of the fetchers compute "lagged" features – e.g. the number of app clicks last week, two weeks ago, three weeks ago, etc. It's worth mentioning that the target variable was obtained with a separate fetcher; in fact the number of days until the actual cancellation was recorded and inside the model I converted it to a "1" if the number of days was inside the upcoming two weeks.

I did a fair amount of exploratory data analysis (EDA) to uncover which features were meaningful from a uni-variate point of view. Of course the usual assumptions that have to be made on distributions in order to use parametric statistical didn't hold. None of the continuous features I obtained were normally distributed, nor did they seem to follow any common distribution I knew off. I ended up using non-parametric tests – e.g. the Mann-Whitney test – but all the effects turned to be significant. The thing is that statistical tests do not reason as human beings do; small deviations in large samples can turn an effect to be significant without it being particularly noticeable from a human perspective. Although using lower p-values can mitigate this effect, in the end I preferred relying on confidence intervals to determined if a feature was "powerful" or not.

I found that doing a "ping-pong" between the model and the feature extraction helped a lot. A lot of machine learning models have some measure of "feature importance" which can be used to determine which features are relevant or not. This helps to look for features in the right direction instead of stabbing in the dark. All in all using statistical tests instead of these feature importances felt like using a proxy which was based on too many theoretical assumptions.

3.4 Scoring metric

The task at hand is a binary classification one. We want to know if a customer will churn in two weeks or less or not. As such the following outcomes can occur when making a prediction:

- True positive (TP): the model says that the customer churns and he actually does, this is a good thing and this is when we want to take preventive action.
- False negative (FN): the model doesn't pick up that a customer will churn, this is bad thing because we could have taken preventive action.
- True negative (TN): the model correctly picks up that a customer will not churn, which is good because we don't want to take preventive action when it isn't need, obviously for cost saving but also to avoid "annoying" the customer.
- False positive (FP): the model tells us that a customer will churn when in fact he doesn't, this isn't good because it means we will be taking preventive action when it isn't needed.

To avoid making too many bad mistakes, our goal is to keep positive precision score ($\frac{TP}{TP+FP}$) above a certain threshold. For example if we say that out of 100 times the models says that a customer will churn it is correct 95 times then only 5 preventive actions could have been avoided.

We also want to be good at catching churners, which translates to positive recall ($\frac{TP}{TP+FN}$). It's OK to make precise calls but we also to catch a lot of the churn that actually occurs. For example if there 10000 customers who churn and that we have a 95% score then this could imply we correctly detect churn 95 times out of 100 but 9500 out of 10000!

In business other words we want to detect the most cancellations possible whilst not predicting that subscriptions will cancel when it fact they will not. In data science words we want to maximise the positive precision and the positive recall. To compare models we can use a single score which is the F1 score ($\frac{2 \times precision \times recall}{precision + recall}$). However, because the output is going to translate to preventive action then a human has to decide what thresholds to use.

Out of the box a machine learning considers a probability above 0.5 as being a "1". However, although this might seem to be the point at which the F1 score will be maximised it might lead to a over-zealous model or a too timid one. Having a human intervention to A/B test the output probability and decide what threshold to use based on cost is necessary and desired by stake-holders. All we can do as data scientists is to provide better performing models across the board so as to offer better compromises. In the end I found that maximising the ROC AUC score proved to be the best alternative because what matters in the end is how good we are at ranking customers between each other. Indeed in the end CRM might want to operate on a limited budget and take preventive action on the 1000 customers who have the highest probability of churning – even if that probability is not high they are the likeliest to churn.

3.5 Applying Gradient Boosted Machines

Right off the bat I decided to use a GBM for my model. My experience with these difficult problems is that, on average, GBMs will always do reasonably well. Moreover, the task at hand has a very strong class imbalance which GBMs can deal with quite well – at least better than most models. Whats more, there exist a lot of good frameworks that are actively maintained and have good Python bindings.

On a side-note I also tried out neural networks, logistic regression and support vector machines but their performance was always lower than GBMs. I didn't do too much research as to why but as usual GBMs are the best "off-the-shelf" models. Neural networks are very difficult to tune, logistic regression are too simple and have a hard-time picking up on non-linear interactions, finally SVMs simply cannot be trained in reasonable time on large amounts of data – the complexity grows with the number of samples for any kind of kernel.

Using a GBM is also handy for determining feature importance. Indeed simply counting the number of trees a feature is used in or even the number of times a feature is used in a split proves to be a robust way of assessing the predictive power of a feature. However, this doesn't make us able to say phrases such as "If feature x increases by dx then the probability of churning increases by dy", but it nonetheless gives a good idea of what makes the model work or not.

For comparing models and model parameters I built a training set and validation set that were not too far in past – the data went from January 2017 to April 2017. Keeping the data static and using constant seeds for each model is extremely important to producible reproducible results.

Initially I ran an XGBoost model and got some baseline results. Tuning the parameters through grid-searched cross-validation proved to be somewhat useful but I was only able to gain one point of ROC AUC after a lot of work. Adding features and QAing (Quality Assurance) proved to yield more meaningful progress. The fact that the model has to be retrained each week means that optimal parameters for one week are not necessarily optimal for the week afterwards. Upon realising this I decided to not spend a lot of time tuning the model; instead I wrote some code to automate the process and to run a grid-search across a wide range of parameters, the idea being that each week the model would find new better parameters based on the latest training data available.

One big mistake I made occurred when I was building my pipeline to include class re-balancing. After doing some research I realised that using oversampling and/or undersampling can help machine learning algorithms learn unbalanced datasets. Oversampling procedures such as Synthetic Minority Over-sampling Technique (SMOTE) [2] generate new instances by sampling from a random distribution fitted on the existing data. Undersampling techniques remove instances whilst preserving the initial distribution. My mistake was applying re-balancing *before* the cross-validation phase. The issue is that by doing so synthetic instances are introduced in the validation set that are very similar to some of the instances in the training set, which results in biased results which are over-confident. Re-balancing has to be applied separately to each fold during the cross-validation, not before.

After a few runs I found that right off the bat oversampling was performing better than undersampling, which seems quite a common result. Instinctively I thought that having balanced labels would lead to a parsimonious model. However, I noticed that if the oversampling ratio was too high then a lot of the instances synthetically produced would not make a lot of sense and would actually deteriorate performance. I found that performing grid-searched cross-validation to search for the optimal oversampling ratio significantly improved the performance of the model, which isn't a practice I came across whilst doing my research.

Framework-wise I used XGBoost for quite some time but then I switched to LightGBM which became somewhat popular doing my internship. I used successfully for a personal Kaggle competition and decided to introduce to the data science team at HelloFresh. Quite surprisingly LightGBM was around 30 times faster than XGBoost! Because of the nature of the algorithm, boosting in most of it's forms can't be applied in parallel. The trick is that LightGBM grows each tree in parallel. In other each split in the tree is assigned to a separate thread. Effectively the deeper the tree the more gain there is to be made. Moreover LightGBM allows to grow unbalanced trees with different depths for each tree. In practice this translates to a `num_leaves` parameter which determines how many leaves are allowed in total, which contrasts with the more common `max_depth` parameter used by XGBoost. One can also mimic XGBoost's behavior with LightGBM by using the `max_depth` parameter and using 2^{max_depth} for the `num_leaves` parameter. All in all the performance between LightGBM and XGBoost was quite similar but the speed gain obtained by using LightGBM was so good that I decided to use it, especially considering the fact that we only had less than a day for training each week. A short time before the end of my internship Yandex released their own GBM framework called CatBoost, sadly I didn't have time to compare it with XGBoost and LightGBM but it seems promising.

Finally the last step of my pipeline consisting in *calibrating probabilities*. A lot of machine learning give poor estimates of probabilities, usually they are too confident in their predictions and return probabilities that are very close to 1 or to 0 but never to, say, 0.5 – think sigmoidal curves. Logistic regression naturally returns well-calibrated probabilities because it directly minimises log-loss. However ensemble methods such as GBMs and Random Forests tend to return probabilities whose distribution peaks around 0.1 and 0.9. This is even more the case for SVM models. Two approaches for performing calibration of probabilistic predictions are provided:

- A parametric approach based on Platt's sigmoid model – commonly referred as "Platt scaling" – which does nothing more than train a logistic regression on top of the output of a model.
- A non-parametric approach based on isotonic regression. The literature seems to say that this is the best way to go if a lot of data is available.

Isotonic regression was new to me so I can't really say why it works better than Platt scaling, in the end I simply added the calibration method as a parameter to the grid-search.

3.6 Results

Considering the fact that the features I used were not very advanced and didn't include much, if not any, user interaction with the website the model did reasonably in the end. Particularly, with the addition of over-sampling with an optimised ratio the model became very good at ranking customers, which can be seen on the following ROC curve. At the beginning of my internship I was getting ROC AUCs of around 0.82 by using XGBoost "off-the-shelf" so to say, obtaining a 0.11 score improvement simply by using oversampling and performing grid-search was quite satisfactory. My feeling that there isn't much more that can be done from a model perspective, instead further work should be put into obtaining more sophisticated features.

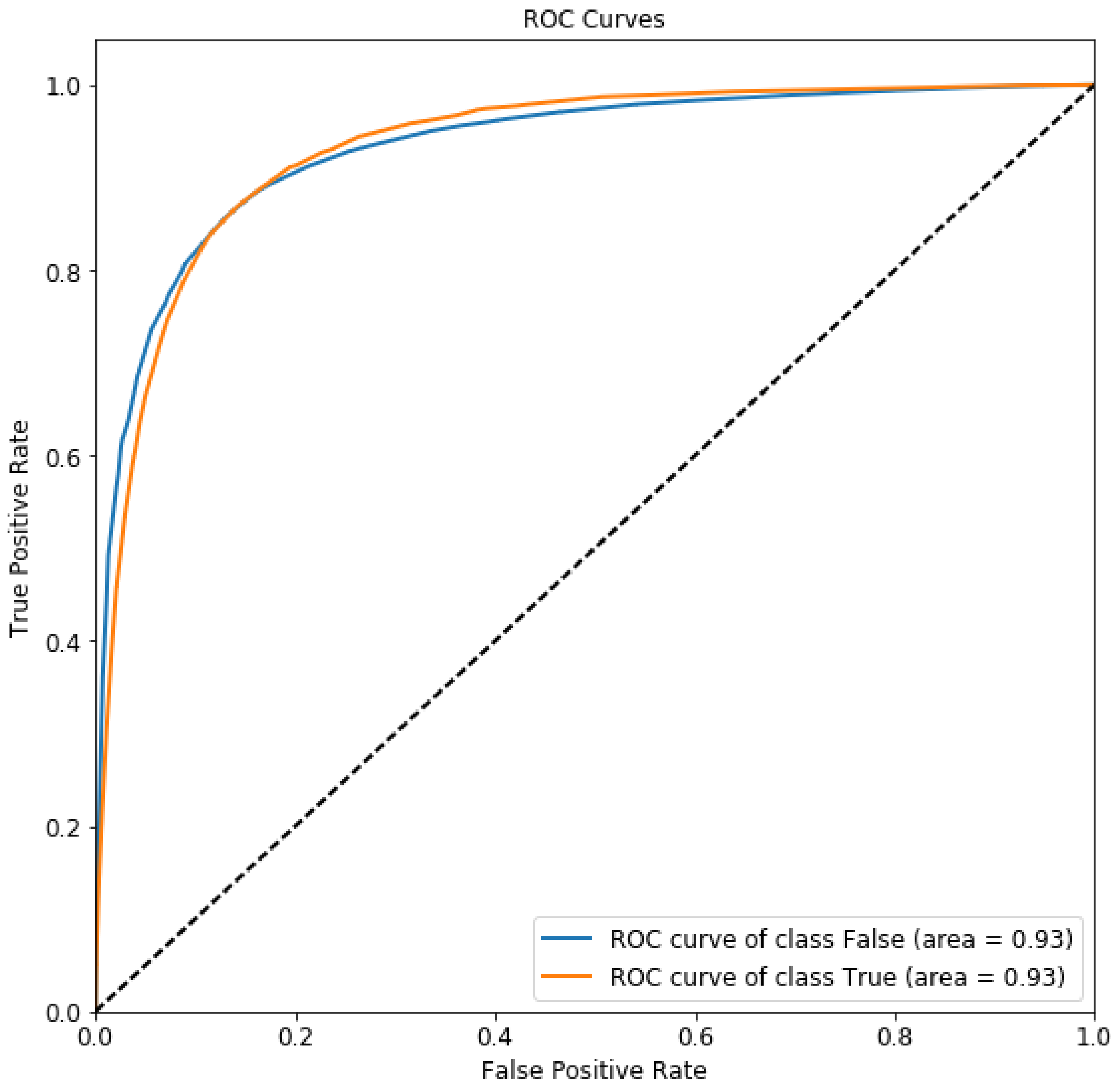


Figure 3.3: ROC curve obtained for the best model

Although the model was quite good at ranking, it didn't do as well at being very precise whilst maintaining recall, as can be seen on the following chart. Basically if we wanted to make sure that 75% of our churn calls were correct then we would only be catching 10% of all churners each week. In a sense this is fine because CRM are working with a limited budget and only want to target a set number of people, for which purpose the ROC score is more appropriate. However the low precision-recall AUC for the positive class shows how difficult of a task this is and means that there still is a lot of work to be done – the best precision-call AUC I got was 0.523, which is very close to 0.5 which is the baseline one would obtain by flipping a coin.

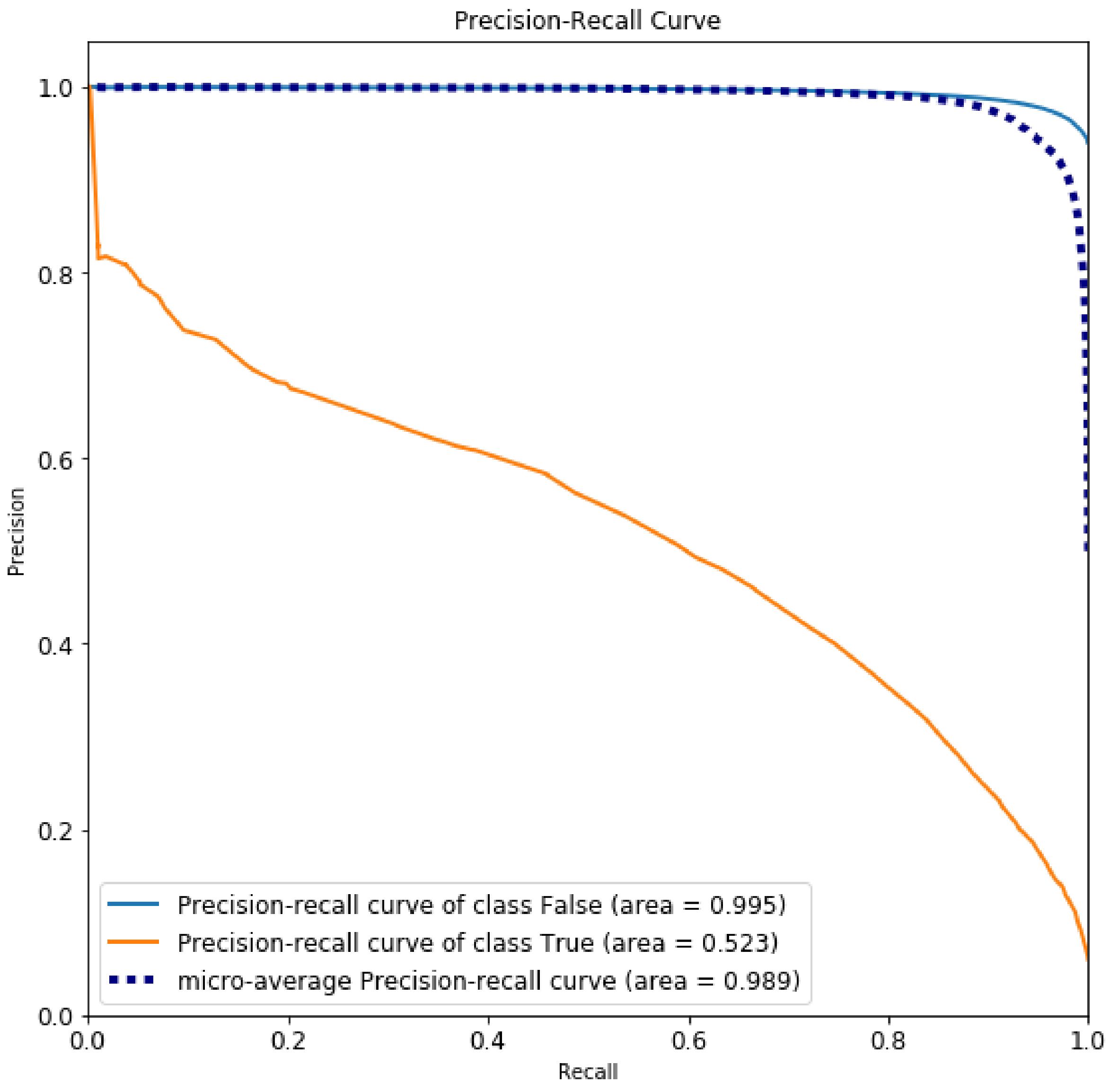
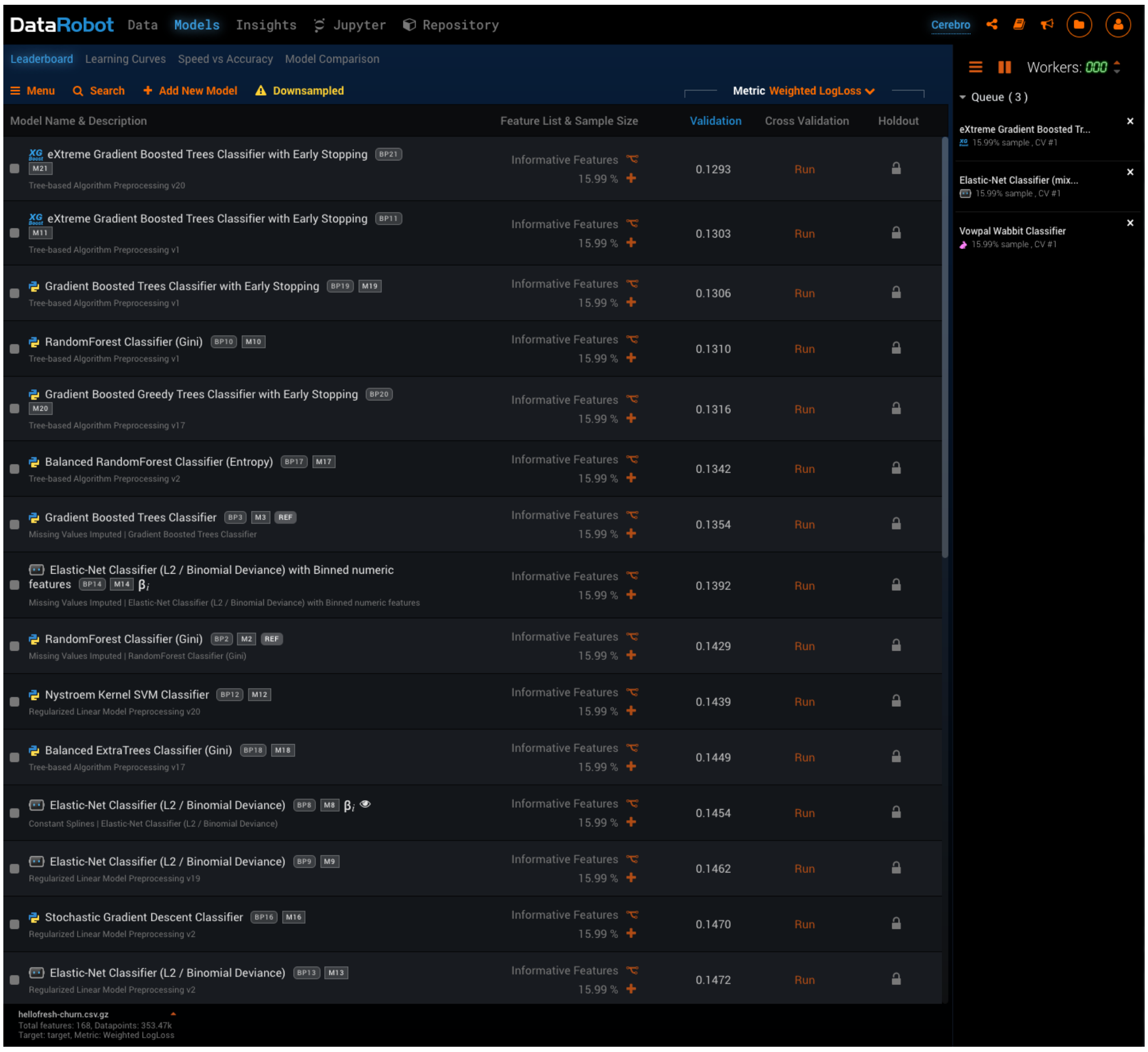


Figure 3.4: Precision-recall curve for the best model

3.7 DataRobot - Automated machine learning

As many companies we get approached by B2B companies who want us to pay to their products. There exist a few niche companies who specialise in helping companies data science tasks. One of these companies, DataRobot, approached us with their automated machine learning application. Although in the end we didn't subscribe to their offer it was enjoyable to use it and to see what, potentially, the future of applied machine learning might be made off.

Basically DataRobot applies a long list of models to a dataset provided by the user and returns the best one. The gain is that it automates so much tedious and error-prone work that is usually grudgingly done by data scientists. The following image shows a list of models before they are applied to a dataset.



Model Name & Description	Feature List & Sample Size	Validation	Cross Validation	Holdout
eXtreme Gradient Boosted Trees Classifier with Early Stopping BP21 M21 — Tree-based Algorithm Preprocessing v20	Informative Features 15.99 %	0.1293	Run	🔒
eXtreme Gradient Boosted Trees Classifier with Early Stopping BP11 M11 — Tree-based Algorithm Preprocessing v1	Informative Features 15.99 %	0.1303	Run	🔒
Gradient Boosted Trees Classifier with Early Stopping BP19 M19 — Tree-based Algorithm Preprocessing v1	Informative Features 15.99 %	0.1306	Run	🔒
RandomForest Classifier (Gini) BP10 M10 — Tree-based Algorithm Preprocessing v1	Informative Features 15.99 %	0.1310	Run	🔒
Gradient Boosted Greedy Trees Classifier with Early Stopping BP20 M20 — Tree-based Algorithm Preprocessing v17	Informative Features 15.99 %	0.1316	Run	🔒
Balanced RandomForest Classifier (Entropy) BP17 M17 — Tree-based Algorithm Preprocessing v2	Informative Features 15.99 %	0.1342	Run	🔒
Gradient Boosted Trees Classifier BP3 M3 REF — Missing Values Imputed | Gradient Boosted Trees Classifier	Informative Features 15.99 %	0.1354	Run	🔒
Elastic-Net Classifier (L2 / Binomial Deviance) with Binned numeric features BP14 M14 β_i — Missing Values Imputed | Elastic-Net Classifier (L2 / Binomial Deviance) with Binned numeric features	Informative Features 15.99 %	0.1392	Run	🔒
RandomForest Classifier (Gini) BP2 M2 REF — Missing Values Imputed | RandomForest Classifier (Gini)	Informative Features 15.99 %	0.1429	Run	🔒
Nystroem Kernel SVM Classifier BP12 M12 — Regularized Linear Model Preprocessing v20	Informative Features 15.99 %	0.1439	Run	🔒
Balanced ExtraTrees Classifier (Gini) BP18 M18 — Tree-based Algorithm Preprocessing v17	Informative Features 15.99 %	0.1449	Run	🔒
Elastic-Net Classifier (L2 / Binomial Deviance) BP8 M8 β_i — Constant Splines | Elastic-Net Classifier (L2 / Binomial Deviance)	Informative Features 15.99 %	0.1454	Run	🔒
Elastic-Net Classifier (L2 / Binomial Deviance) BP9 M9 — Regularized Linear Model Preprocessing v19	Informative Features 15.99 %	0.1462	Run	🔒
Stochastic Gradient Descent Classifier BP16 M16 — Regularized Linear Model Preprocessing v2	Informative Features 15.99 %	0.1470	Run	🔒
Elastic-Net Classifier (L2 / Binomial Deviance) BP13 M13 — Regularized Linear Model Preprocessing v2	Informative Features 15.99 %	0.1472	Run	🔒

Queue (3)

eXtreme Gradient Boosted Tr...
15.99% sample , CV #1

Elastic-Net Classifier (mix...
15.99% sample , CV #1

Vowpal Wabbit Classifier
15.99% sample , CV #1

hellofresh-churn.csv.gz
Total features: 168, Datapoints: 353.47k
Target: target, Metric: Weighted LogLoss

Figure 3.5: DataRobot interface

At it's core DataRobot has a list of blueprint "blueprints" which are nothing more than machine learning pipelines that can plugged into any valid dataset. Some of these blueprints are proprietary but most simply boil down to applying implementations that come from popular libraries such as scikit-learn and H2O.

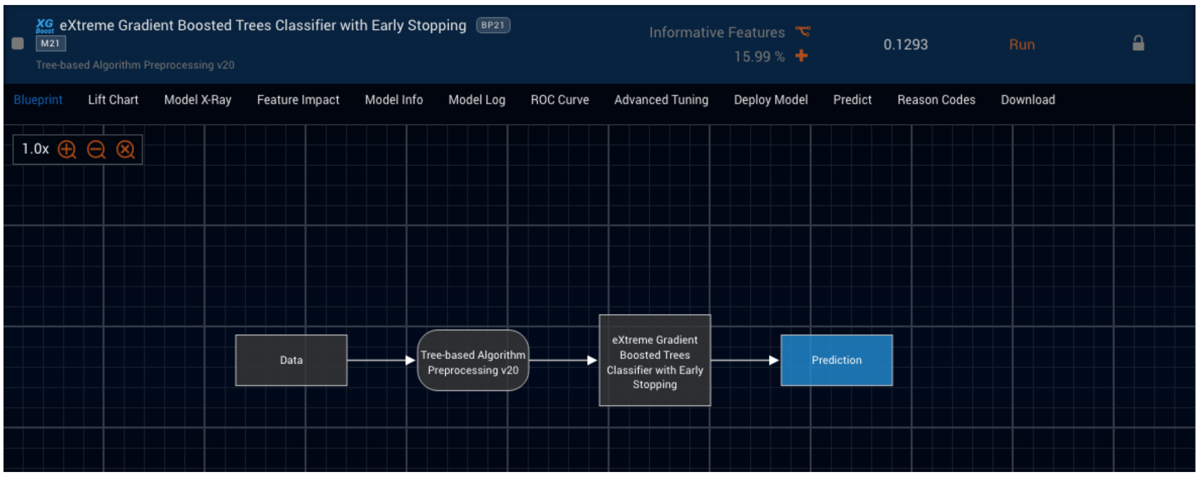

eXtreme Gradient Boosted Trees Classifier with Early Stopping
BP21
M21
Tree-based Algorithm Preprocessing v20
Informative Features
15.99 %
0.1293
Run
Blueprint
Lift Chart
Model X-Ray
Feature Impact
Model Info
Model Log
ROC Curve
Advanced Tuning
Deploy Model
Predict
Reason Codes
Download
1.0x
Data
Tree-based Algorithm Preprocessing v20
eXtreme Gradient Boosted Trees Classifier with Early Stopping
Prediction

Figure 3.6: DataRobot model blueprint

At it's simplest DataRobot loops through dozens of models and trains/tests them on similar on identical training and test sets. Of course the gain for the data scientist is that all this is battle-tested and can all be done without having to write any code. The user can choose what metrics he/she wishes to optimise. A nice feature is that the user can decide how the cross-validation procedure should be done, which also includes performing temporal cross-validation as can be seen in the following image.

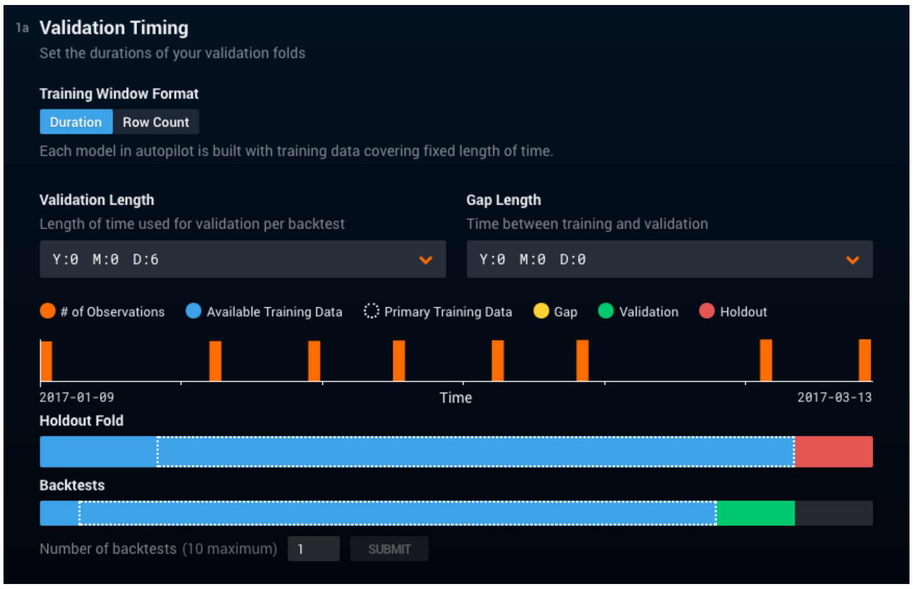
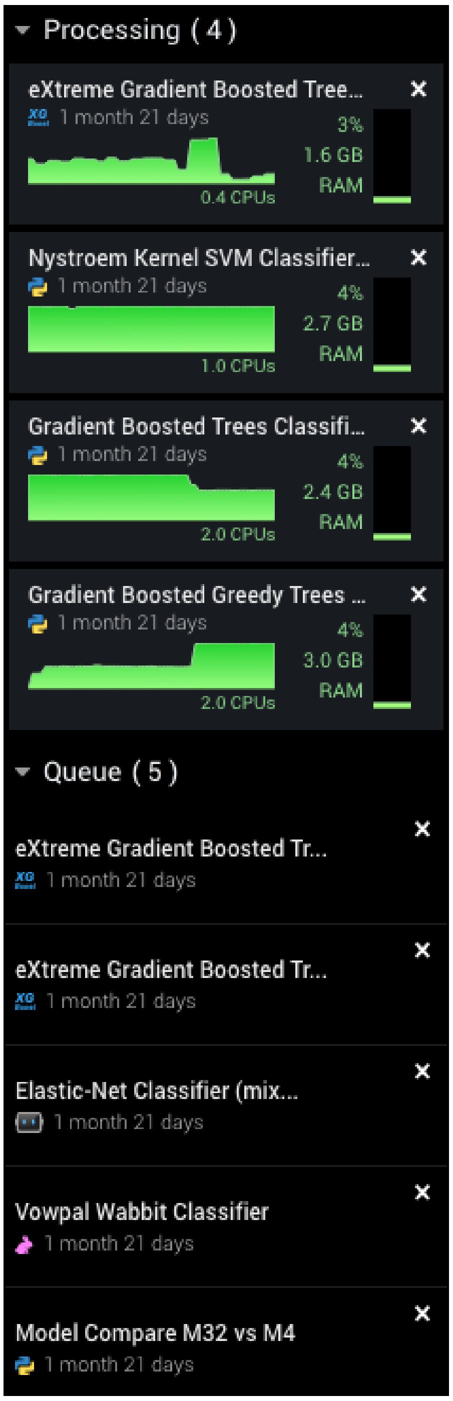

1a
Validation Timing
Set the durations of your validation folds
Training Window Format
Duration
Row Count
Each model in autopilot is built with training data covering fixed length of time.
Validation Length
Length of time used for validation per backtest
Y:0 M:0 D:6
Gap Length
Time between training and validation
Y:0 M:0 D:0
● # of Observations
● Available Training Data
Primary Training Data
● Gap
● Validation
● Holdout
2017-01-09
Time
2017-03-13
Holdout Fold
Backtests
Number of backtests (10 maximum)
1
SUBMIT

Figure 3.7: DataRobot data partitioning

Based on the tier that has been purchased, more or less "workers" can be allocated to train the models. For example in the following four workers are running models concurrently. A nice feature is that one may implement a given interface and use a custom model. In that case DataRobot is just a fancy – and expensive – software for running the model, it nonetheless provides a nice framework to compare models between each other.



Processing (4)

eXtreme Gradient Boosted Tree...
1 month 21 days
3%
1.6 GB
RAM
0.4 CPUs

Nystroem Kernel SVM Classifier...
1 month 21 days
4%
2.7 GB
RAM
1.0 CPUs

Gradient Boosted Trees Classifi...
1 month 21 days
4%
2.4 GB
RAM
2.0 CPUs

Gradient Boosted Greedy Trees ...
1 month 21 days
4%
3.0 GB
RAM
2.0 CPUs

Queue (5)

eXtreme Gradient Boosted Tr...
1 month 21 days

eXtreme Gradient Boosted Tr...
1 month 21 days

Elastic-Net Classifier (mix...
1 month 21 days

Vowpal Wabbit Classifier
1 month 21 days

Model Compare M32 vs M4
1 month 21 days

Figure 3.8: DataRobot model evaluation

In the end our needs – and our budget – didn't match with DataRobot and we didn't pursue with them. However, we got to run it with the same data I used for training a GBM and I managed to beat it! My luck was that being fairly new DataRobot doesn't include "fancy" tricks such as re-balancing so I was able to beat it by using oversampling. Regardless it was lifting to see that we doing as could as we could given the features that we had. This again confirms my feeling that there isn't much more valuable work to be done on the side of the model, but instead time should be spent fetching new features.

3.8 Deployment

The deployment of a machine learning application was totally new team, moreover this was the first one that was written in Python at HelloFresh so I didn't much guidance. Luckily once I got to the deployment phase the data science team was starting to use AWS so I was allowed to use a separate server for my so-called "churn model".

3.8.1 Integration with Cerebro

Because the data collecting was done through Cerebro we decided to have two separate servers, one for Cerebro and one for the churn model. The idea was that Cerebro would collect the data on a regular basis and store in an Elastic File System (EFS). An EFS is a file-system provided by AWS that can be mounted on different servers so that these servers can share information. This is extremely practical because it allows application to be stateless because everything they consume and produce is stored separately, this way we don't have to worry about data being lost if a server stops or crashes – which can happen on AWS!

3.8.2 The doit library

To organise my code I decided to think of my model in it's entirety as a directed acyclic graph (DAG) where each step of the model consumed inputs and produced outputs. For example one of the steps of the models queried the HDF store maintained by Cerebro and created training and validation sets. Another step performed the cross-validation by taking as input training/validation folds and outputting a JSON file with optimal parameters.

At first I considered one of the popular frameworks such as Luigi or Airflow but I realised they didn't suit my needs, mostly because they had too many bells and whistles. I also thought about using Makefiles, however they are a bit clumsy when multiple inputs are required. In the end I settled on using a not so popular Python library called "doit". With doit one may define a list of steps in Python where one simply has to describe what are the inputs, outputs, and intermediary actions; doit will then take care of determining what steps to execute and in what order. Moreover, doit can be launched from the command-line so it is very simple to run it from a crontab file. I thoroughly recommend and it has sparked an interest in me to formalise data pipelines.

Chapter 4

Building a basic recommendation engine

Personalization and experiences tailored customer are becoming standard in many industries. One of HelloFresh's adversaries in the United States, namely Blue Apron, recommends recipes that are based on the dietary preferences each customer provides them. Specifically, each customer can indicate which protein he/she usually eats. However, what Blue Apron is not truly personalization per say, indeed they are recommending recipes out of the ones they propose at a given week. They are not customising the recipes to each customer, for example by adding or removing the amount of spice in the food box. Instead, Blue Apron makes sure that if a customer does not actively choose recipes on the website, then at least he/she will receive recipes in his/her box that suit at best his dietary preferences.

In June a group of people at HelloFresh started to discuss how personalization should be done. One of the goals was to catch up with what Blue Apron had done by the end of July, more precisely to build an MVP on which further improvements can be based on. Putting into place a recommendation requires efforts from multiple teams, from the UI/UX team who has to build a page where customers can modify their preferences to supply chain management (SCM) who have to be even more flexible when forecasting demand. The role of the data science team is to provide a recommendation engine that recommends appropriate meals based on customer preferences.

The MVP targets customers from the United States who have just registered on the website. Upon registration these users will be shown a form where they can select what proteins they like or dislike. There are 6 indicated proteins (beef, fish, lamb, pork, poultry and shellfish), which makes a total of $2^6 = 64$ combinations. Because at registration we virtually know nothing else about a customer, at least concerning his/her tastes, it seems a reasonable assumption to use the preference combination he/she indicates. In other words we can consider making recommendations for one preference combination instead of making recommendations for each customer because as far as we know it is the same thing. As will be discussed in the last section, the method that we built can be extended to existing customers without too much difficulty.

There are some specificities of the meal-kit industry that have to be addressed in order to build a recommendation engine:

1. Each week there are only around 8 recipes we can recommend from, this isn't a text book scenario
2. Recipes change on a weekly basis, thus some generalisation is required in order to infer what people like
3. It is not said that a person who only likes fish can be satisfied, indeed in the US there are only 1 or 2 meals a week that contain fish

Usually the goal of a recommendation engine is to select relevant items from a somewhat large set of items. From this large set we usually expect there to be at least a few, if not many, items that are relevant to a particular user. In our case we want to recommend 3 recipes, by simply filtering recipes for a given week based that they contain a given protein or not can lead to one of the following scenarios:

1. Exactly 3 recipes match, which means that no further work has to be done to provide relevant recipes.
2. Less than 3 recipes suit the preferences, in which case we have to compromise and minimise the bad experience.
3. More than 3 recipes suit the preferences, in which case we have to maximise the customer's satisfaction by choosing the best 3 recipes.

At the moment users who have not chosen recipes for a given week get the first 3 recipes of the week which is a global default. The goal of this project to replace this default choice by something a bit smarter which, even if it can't transform beef into fish, will nonetheless provide an optimal recommendation based on what is possible.

The goal is to build a recommendation engine which handles all 3 scenarios transparently. In case not enough recipes match a customer's preferences the engine should provide "least worst" results. The next section gives an outline of the approach I developed and which got used for the MVP.

4.1 Nearest neighbours approach

One way of going would be to write a lot of `if` statements and to describe manually how to proceed in each scenario. For example if only a single fish dish is available and a user specifically said that he only likes fish then we can recommend him chicken because it is less offensive than pork. However possible, it's easy to see that this approach does not scale and possibly requires a lot of maintenance.

We can instead rely on what customer actually and create some sort of self-supporting system. Intuitively, if people who are similar to you have a choice for next week, then we can use that choice for making you a recommendation. In the case of customers who only eat fish, customers compromise naturally when choosing recipes by considering what they "dislike the least". By considering many people like you we can start making a good guess a what we should recommend to you.

In machine learning this process is called and is one of the simplest and most intuitive algorithms in the field. In ML the k-nearest neighbours (KNN) is a supervised method that can be used for both classification and regression, it is also one of the first algorithms a student learns about because of it's simplicity and intuitively. In computer science nearest neighbours search is used in an unsupervised way for many applications. Data structures involving trees – typically ball trees and kd-trees – can be used to speed up the search without losing much accuracy.

In our case, the main difficulty lies in determining a similarity measure between new customers and ones who have made a choice for next week. Indeed for new customers we only know what protein preferences they have provided us with, in practice their preferences can be stored in a vector - for example $(0, 1, 0, 0, 0, 0)$ means that the customer only eats fish. Existing customers have not filled out the survey and even if we sent it to them not all of them would fill it out. However a lot of existing customers have made recipe choices in the past; what we can do is "guess" how they would have filled out the survey based on what proteins they chose.

Say we know which customers have chosen next week and for those customers we know what they have chosen in the past, this results in the following table.

Customer ID	Recipe
1	Milanese veal
1	Tenderloins
1	Steak and chips
2	Chicken tikka
2	Chicken skewers
2	Roasted lamb
3	Fish and chips
3	Fish burger
3	Fried calamari

Each recipe HelloFresh has ever offered contains ingredients that are manually tagged in a back-office application. With this information and with the ingredient information recipe it's quite easy to assign a protein to each recipe. Some recipes contain more than one protein – for example a chicken burger which contains some bacon – but in the end it doesn't matter. Indeed with the information available it's possible the following table which shows how many times a protein was offered to each customer and how many times that customer chose it. In the United States a lot of the recipes contain beef which will mechanically make beef seem more popular than other proteins, however by considering how many times each protein is offered gives a less skewed idea of how much a customer likes each protein. For example a user who could have chosen fish 3 times and actually chose fish each time probably likes fish. On the contrary somebody who chose beef 17 times out of a possible 34 doesn't necessarily like beef but chose it because *there was no other choice*.

Customer ID	Protein	Times offered	Times chosen	Ratio
1	Beef	34	17	0.5
1	Fish	3	3	1
1	Lamb	1	0	0
1	Pork	5	4	0.8
1	Poultry	25	20	0.8
1	Shellfish	2	2	1

We can use these “protein ratios” as proxies for the survey answers and thus determine the nearest neighbours of each preference combination. For example the beef and poultry preference combination of vector $(1, 0, 0, 0, 1, 0)$ will have neighbours who have beef and poultry ratios close to 1 whist their other protein ratios will be close to 0.

Once we have found “neighbours” for a specific preference combination, we have to do some voting procedure to decide what recipes to recommend from the ones the neighbours chose. Indeed, neighbours have not necessarily chosen the same recipes and some agreement has to be reached. This paradigm is very similar to the KNN algorithm where each neighbour contributes to the classification/regression result with some voting power.

We can simply count the number of times each recipe was chosen by all the customers and recommend the ones with the highest counts. In other words each customer has an equal say in the voting procedure. Another way to proceed is to assign voting power to each power based on how close the customer is to the preference combination. It makes sense that a customer with a vector of $(0.8, 0, 0, 0, 0.9, 0)$ should be a higher say than a customer with a vector of $(0.75, 0, 0, 0, 0.85, 0)$ if we are making recommendations for the preference combination of vector $(1, 0, 0, 0, 1, 0)$. Indeed by using the Euclidean distance we can use the resulting distance to weight each vote.

This “weighted voting procedure” solves a typical problem: how to choose the number of neighbours to involve. For the KNN algorithm this parameter – usually denoted k – is usually determined by grid-searched cross-validation. This is of course recommended and should be done in an experimental setting. However weighting the votes brings stability to the voting process and can in effect require using less customers than in the “non-weighted voting procedure”. The issue is that it’s very that by including more neighbours the recommendations will get skewed because the neighbours that are furthest away have nothing to do with the preference combination. In practice it’s possible that most people indicate that they eat most proteins so including more or less neighbours won’t change the results much. However, for smaller groups – for example fish eaters – including more neighbours might entail including chicken or pork eaters which will result in wrong recommendations. By weighting the votes we can make sure that neighbours who are too far away from the preference combination have a contribution to vote that is meaningless compared to “good” neighbours.

The following chart shows how the voting procedure turns out with equal say. The y-axis indicates the recipe number for the upcoming week (here there are 13 but some of them are disabled) whilst the x-axis shows the number of neighbours that are considered. Each line stands for a position in the ranking – the blue line being the pole position recipe – which in others means that it's the recipe that got the highest number of votes. Visibly the ranking changes quite a lot as the number of neighbours fluctuates, for example the blue line and the orange line keep swapping with each other quite often. However, after a 100 neighbours there doesn't seem to be much instability – apart from the red line and purple line but then again they don't represent the top 3 chosen recipes so it doesn't affect the final recommendation.

The fact that lines exchange frequently typically means that there is a tie between two recipes. A tie between two recipes can mean that the neighbours do not agree on what is there favourite recipe and probably pick default or to compromise. There are a few reasons why these ties can occur. On the one hand a group of chicken lovers might always pick two chicken recipes if they are available – in fact this is the case with the chart below – which results in there being no clear "best" recipe. On the other hand fish lovers might agree on the pole position but might have different coping mechanisms to deal with the fact that no other recipes satisfy their dietary preferences. In effect, customers who don't know what to choose might pick recipes at random which will result in frequently alternating lines. Moreover, unsatisfied users might pick recipes based on how the recipe pictures appeals to them – and they might not all have the same visual tastes – which again results in frequently alternating lines.

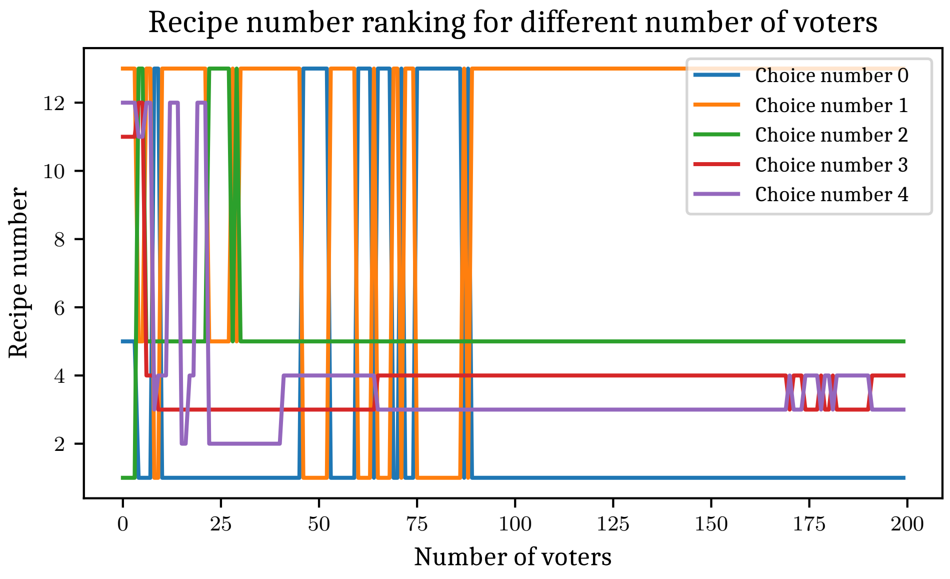


Figure 4.1: Recipe ranking with non-weighted voters

We could decide to use a lot of neighbours to gain in stability, but then again for small segments this might negatively affect the final recommendation. Moreover, deciding on a different number of neighbours per preference combination does not scale and is difficult to maintain. The following chart shows how the voting procedure turns out with weighted votes. Each customer can vote with a weight of $\frac{1}{d_i}$ where d_i stands for the distance to the customer's preference combination.

Quite clearly the weighted procedure results in lines that alternate less frequently, in other words the voting procedure is more stable. In fact in this particular example the top 3 recipes – represented by the blue, orange, and green lines – do not change after just 5 neighbours with the same results as the ones produced by the non-weighted voting procedure which needed over 90 neighbours to reach stability. In practice, considering 90 neighbours isn't an issue but there are benefits to needing less neighbours. Indeed if we want to change our method and make recommendations at a customer level instead of a segment level then there might be very small clusters of individuals, moreover using neighbours means using less computation power which would help scale the method.

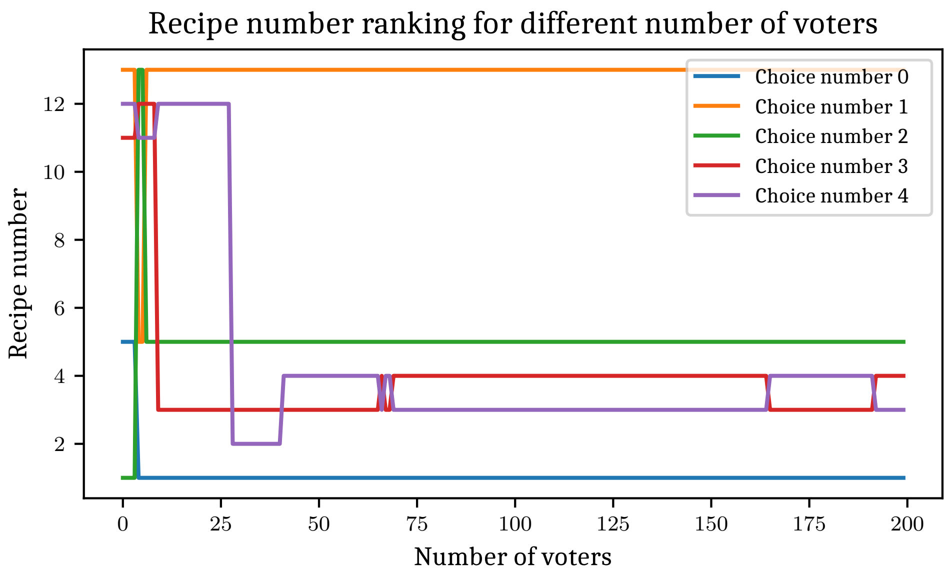


Figure 4.2: Recipe ranking with weighted voters

The last piece of the puzzle lies in deciding when to update the recommendations. As time goes by more and more people choose recipes for the upcoming which can possibly produce different recommendations. For the MVP we settled on updating the recommendations every 4 hours. To avoid bad experiences where a customer might see a recommendation and trust before the recommendation being updated we offer the customer the possibility to "lock in" a recommendation to avoid being updated. However that is the concern of the backend and frontend teams – not the data science team.

4.2 Implementation details

We decided quite naturally to implement the MVP in Python. It offers a wide set of tools that suffice to do just about anything, particularly for data science tasks. The two potential bottlenecks to our algorithms are the fetching of the data and the computation of the nearest neighbours. In the first case the speed at which the data is fetched depend entirely on the underlying database on which we have little influence. As for computing the nearest neighbours the scikit-learn library [5] leverages optimised C code. Regardless, Python is very good for prototyping, is easy to deploy and we are dealing with large amounts of data.

In practice fetching the necessary data from the data warehouse took around 15 minutes whilst computing the nearest neighbours and the weighted vote only took one minute. Because the historical meal choices only change once a week a separate scripts takes care of obtaining the data and storing it in an SQLite database. A second script could then leverage that data and compute the nearest neighbours every four hours or so.

All in all, the final MVP only took around 300 lines of code. I made of point of heavily commenting my code by leveraging Python docstrings and explicitly describing what were the inputs and outputs of the different functions by using Markdown syntax tables.

4.3 Further steps

The MVP performs recommendations for new joiners of whom we only know the protein preferences. Many extensions and scenarios can be addressed. The role of the data scientist is first of all to monitor the results of the MVP recommendation engine and secondly to advise decision-makers in the company on what is feasible and actionable.

One natural extension is to ask for more preferences than only proteins. Indeed other polarising aspects of recipes might include “exotiscim” or spiciness. To extend the registration survey we first have to analyse how customers actually answer it. It might be that 90% of customers indicate that they all proteins – they don’t opt-out of any protein – thus suggesting that proteins are a poor way of distinguishing recipes. On the other hand, if we realise that the customer KPIs we are tracking increase significantly for customer who got affected by the MVP then we might not want to modify it. All in all there is always some decision to be made as to how much the MVP should be extended, especially if it is successful. Sadly the monitoring of the recommendation MVP is due to occur after my internship.

Another possibly strong extension would be to bring the recommendations to a customer level. For this happen we have to leverage past recipe choices, much like we did for the MVP. However, instead of mapping a meal choice history to a protein preference combination we can look for the neighbours of the raw history and perform recommendations per customer, much like Netflix and Spotify operate. Obviously this only works for existing customers who have made meal choices in the past so technically it can beside the MVP which targets new joiners.

Chapter 5

Miscellaneous

As an intern I got to do a lot of mundane tasks. I rather enjoy being a “Swiss Army knife” and working on various topics that data scientists have to deal with on a regular basis. I deemed these tasks worthy of mentioning and I divided them in a few sections.

5.1 Managing the development server

When I arrived our development server was shared by all the data scientists – further on we transitioned towards AWS. RStudio was installed on the server and a lot of the data scientists made full use of all the cores on the servers; to prevent this I assigned cores to each UNIX user to avoid memory hogging. Moreover I took of monitoring processes and ghost daemons to save CPU cycles. I was quite surprised at how some users didn’t realise they were not the only ones using the development machine and that monitoring had to be done. Aside from this I also took care of setting up a Jupyter Hub for Python development. A Jupyter Hub server spins up an individual Jupyter Notebook per user and is quite difficult to put in place.

5.2 Writing small ETL jobs

Although I wasn’t part of all the data science projects – at one point there were more projects than there were team members – I was usually asked to write scripts for transferring data between servers/databases. I also took care of producing reporting spreadsheets and saving them to Dropbox for users to access them. Although this wasn’t very fun work, it was quite satisfying to see these “mini ETLs” run flawlessly every day.

5.3 Batch fraud detection

When I left one of the senior data scientist had just left, soon after the fraud detection he has written crashed and nobody knew how to fix it. The crux of it was that it detect which of the vouchers that were used on the website were frauds or not. For example some customers create new accounts and use a new joined voucher just to get a discount, we consider this fraud and we want to be able to detect it because it is a huge source of loss. I was assigned to building a temporary solution while the IT department worked on a more trust-worthy solution.

What I simply did was to see if I could match a new joiner with an existing cancelled one. For example customers who fraud still have to provide the same address to actually get the box delivered. Naturally cunning customers might changed part of the address so that exact look-up won't work. I quickly realised this and considered using Levenshtein distance to counter this trick, however it proved to be too long to run over all the customers, especially when run in Python and not in a database query. After some research I learnt about the SOUNDEX algorithm that is implemented in MySQL and other popular database systems. Without going into too much detail, the idea is to remove all vowels from a word assign a code to the first few consonants. In the end this works quite well, it is particularly good at detecting fake email addresses where only a few digits are added to the end of the first part of the address – e.g. `maxhalford26@gmail.com` and `maxhalford@gmail.com`. The reason the SOUNDEX algorithm works is that people who fraud usually only change a bit of information without it being completely different, for example they create a fake email address with only a few minor changes because they are lazy.

In practice I wrote a huge SQL query that was run every day and which would compare all the new joiners of the previous day and compare them to the existing customers. For each new joiner I would count the number of matching fields – for example the first name, the last name, the address and the email address. The SOUNDEX algorithm would produce a 1 or a 0 in case of a match or not and I would sum these scores so as to get a global score which would then be used to rank new joiners from most fraudulent to least fraudulent. Customer care would then take this list of customers and check manually if they were frauds or not – they didn't blindly trust my algorithm!

5.4 Scaling the data science team

Because the HelloFresh data science team was relatively new everyone had to put a hand on board to make projects run smoothly. Being proactive was, if not required, essential to being efficient and meeting deadlines.

Concretely I introduced git to the team and explained how it worked conceptually and also in practice by using GitHub. Before most of the code was stored on Dropbox; this sort of worked because each member of the team was working on separate projects. However this isn't good practice and doesn't scale well. Moreover for auditing purposes we are legally obliged to put production code on GitHub with proper code reviewing. Next to this introduction to GitHub I also did a presentation on Docker and how it can be used to deploy data science applications. These two presentations consisted in some slides and a hour of everyone's time during so-called "Data Science Academy" sessions that occurred on Fridays.

Aside from the rest I also helped maintain a data science onboarding document for new joiners. HelloFresh's infrastructure can be quite daunting at first so having some sort of document recapping where to find everything with a FAQ section can be very helpful. Moreover I contributed to building a Google Big Query dictionary to assess what information was made available. This dictionary will prove valuable for building fetchers for Cerebro in the future. Finally I also developed an internal Python package containing utility functions for data scientists and analysts; it helped handling internal time handling and connecting to databases with one-liners.

5.5 Transitioning towards AWS

As I might have already mentioned at the start of my internship all of our work was developed and deployed on internal rack servers. This was fine but it meant we were contrived both in processing power and in extra functionalities. Towards the end of my internship we transitioned towards Amazon Web Services (AWS) which is one of the *de facto* ensemble of cloud tools for running applications. I took the time to migrate Cerebro and the churn model to dedicated AWS servers, during this process I got to use the following AWS tools:

- *EC2 (Elastic Compute Cloud)*: commodity servers that can run general-purpose applications much as one would with any other kind of server.
- *RDS (Amazon Relational Database Service)*: dedicated database servers which can run most of the popular database systems such as MySQL and PostgreSQL. Having the database running on a separate served is strongly recommend so as to decouple the data from the application code.
- *EFS (Elastic File System)*: an EFS can be mounted on multiple so that these servers can share a directory. Much like RDSs, EFSs help decouple data from applications.
- *AMI (Amazon Machine Images)*: this is a very important in the AWS tool chain. The idea is that servers can be "snapshotted" so that they can be reused for different applications. For example I "baked" a AMI that had Python and useful packages installed so that new servers could be spun up using this AMI, instead of starting from scratch each time.

Chapter 6

Conclusion

Overall I had a good time at HelloFresh. Apart from improving my skills in applied machine learning, I also got to understand a lot of business-related concepts and to learn how to bridge the gap between what the business needs and what data science has to offer.

Although I spent a lot putting infrastructure and tooling in place I still got to do some concrete machine learning. It was very enriching to build a machine learning application from A to Z and to see it being used. In general I got a grasp of all the work that has to be done for a data science project to be successful; it's not just about machine learning part and the seemingly complex steps that have to be executed, it's also a lot about communications and understanding the ever-changing needs of the business.

Bibliography

[1] Benjamin Paul Chamberlain et al. “Customer Life Time Value Prediction Using Embeddings”. In: (2017). eprint: `arXiv:1703.02596`.

[2] Nitesh V. Chawla et al. “SMOTE: synthetic minority over-sampling technique”. In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[3] HDF Group et al. *Hierarchical data format version 5, 2000–2010*. 2010.

[4] Li Erran Li et al. “Scaling Machine Learning as a Service”. In: *International Conference on Predictive Applications and APIs*. 2017, pp. 14–29.

[5] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.