Special Problems Final Report

Smart Task Management Web Application Leveraging Supervised Learning

Roshan Prabhakar (roprabha@calpoly.edu)
Advisor: Franz J. Kurfess
December 12th, 2018

Department of Computer Science and Software Engineering,
California Polytechnic State University, San Luis Obispo

Abstract

A variety of different applications have entered the market with their core motivation to use the user's data to provide better insights about the user's behavior by identifying patterns in their data. While this technique has been applied widely to solve optimization problems in fields like finance and transportation, it hasn't realized its full potential to optimize human productivity. We as humans are constantly generating signals regarding our productivity when we use a task management applications such as our calendars, checklists and reminders. In this paper I discuss my journey trying to build an application that incorporates these ideas by allowing users to gain insight into their actions through data visualizations and by using common machine learning techniques to predict user productivity.

Contents

Motivation and Background

Recommendation systems are used by applications such as Youtube (video), Spotify (music) and Amazon (shopping) and countless others to make recommendations based on data that the user generates on their platform to increase metrics such as engagement, money spent or even time spent. While these systems do make useful recommendations, it can be they often get in the way of being productive. According to comScore's 2017 Cross Platform Future in Focus report, the average American adult spends 2 hours, 51 minutes on their smartphone every day.

There is a need to spend our time more efficiently. I wanted to use these same techniques of using data to instead use our time more productively. With rapid improvements in server costs, high level libraries for the web, database management and machine learning, it is a good time to building such a system fairly efficiently to test my hypothesis.

The system will be data driven, i.e. the application will log all the events a user produces. An inherent flaw in the system would be that it would only work well for people who have a daily or weekly routine. If the user's schedule is unpredictable, the model wouldn't perform very well due to the lack of a pattern. But since most of our lives is spent working and living in the same location, this is an edge case.

Requirements

CLIENT - WEB INTERFACE

The system will provide a user interface that allows the user to perform a wide range of task management actions and a well defined organizational structure similar to popular productivity applications. The application will also have a glanceable chart with the predicted productivity score for the user throughout the day.

API SERVER

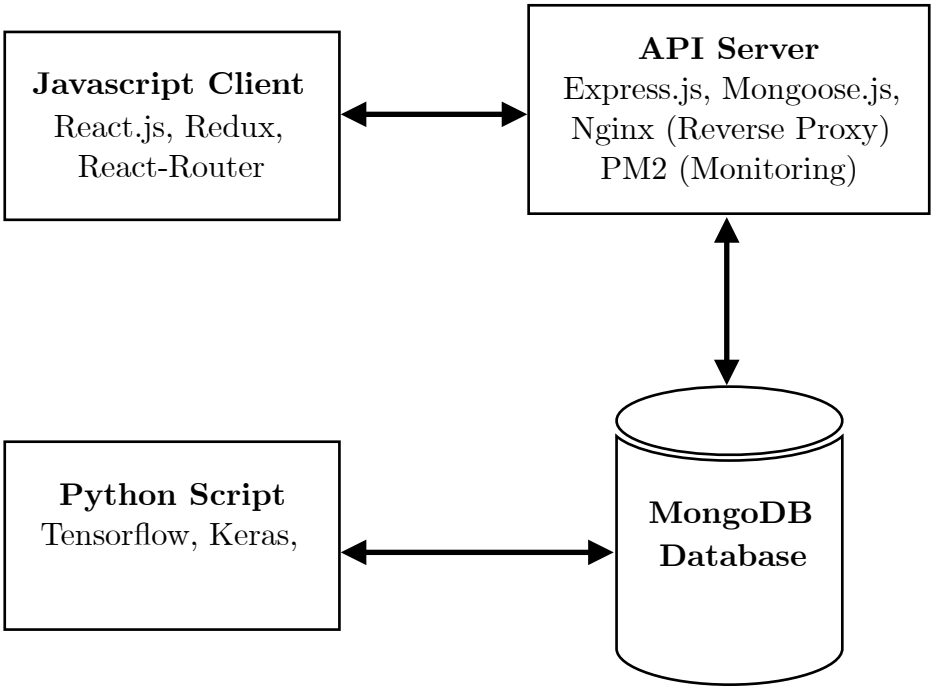
The server will need to communicate with our client over https. The web server will expose numerous endpoints for the client to hit and update the databases accordingly. The other part of this system will be user authentication and authorization.

MACHINE LEARNING RELATED SOFTWARE:

Perform exploratory data analysis on past data to test different supervised learning techniques. Design a system that updates our databases as new task events come in from the user.

Learning Objectives:
The Project requires the student to learn about:
- State management in the browser for client applications (Redux)
- User interface development for the web with React.js
- User authentication techniques
- Setting up web infrastructure to deploy the application
- UI/UX design using Sketch
- Designing APIs around the REST Protocol
- Webserver developement using Express.js
- Modeling schemas for non-relational databases (MongoDB, Mongoose.js)
- Using Keras with a TensorFlow backend to train our model

System Architecture



Client

The application was designed using Sketch which is a user interface prototyping tool. I went through 3 different versions before landing on a unified design language.

The Visual Design Process:

1. Drawing on visual design traits from existing popular task management applications such as Things 3 on iOS, Todoist and Asana.
2. Building the user flow through the application. This included designing the login screen, the dashboard, the sidebar and every task interaction such as add, delete and edit. This job was made easier by designing a design system with specifications for fonts, colors, buttons, dropdown etc.

The two main technologies that power our client will be React and Redux.

Why React?

Here are a few reasons why React has become so popular so quickly:

- Working with the DOM API is hard. React gives us the ability to work with a virtual browser that is more friendly than the real browser. React's virtual browser acts like an agent between the developer and the real browser.
- React enables us to declaratively describe their User Interfaces and model the state of those interfaces. This means instead of coming up with steps to describe transactions on interfaces, we just describe the interfaces in terms of a final state (like a function). When transactions happen to that state, React takes care of updating the User Interfaces based on that.
- React is just JavaScript, there is a very small API to learn, just a few functions and how to use them.

React has three main design concepts that drive its popularity:

1. The use of reusable, composable, and stateful components

In React, we describe User Interfaces using components. Components can be explained as simple functions (in any programming language). We call functions with some input and they give us some output. We can reuse functions as needed and compose bigger functions from smaller ones.
Components are exactly the same; we call their input "properties" and "state", and a component output is a description of a User Interface (which is similar to HTML for

browsers). We can reuse a single component in multiple User Interfaces, and components can contain other components.
Unlike pure functions however, a full React component can have a private state to hold data that may change over time.

2. The nature of reactive updates

React's name is the simple explanation for this concept. When the state of a component (the input) changes, the User Interface it represents (the output) changes as well. This change in the description of the User Interface has to be reflected in the device we're working with.

In a browser, we need to regenerate the HTML views in the Document Object Model (DOM). With React, we do not need to worry about *how* to reflect these changes, or even manage *when* to take changes to the browser; React will simply *react* to the state changes and automatically update the DOM when needed.

3. The virtual representation of views in memory

With React, we write HTML using JavaScript. We rely on the power of JavaScript to generate HTML that depends on some data, rather than enhancing HTML to make it work with that data. Enhancing HTML is what other JavaScript frameworks usually do. For example, Angular extends HTML with features like loops, conditionals, and others. When we receive just the data from the server (in the background, with AJAX), we need something more than HTML to work with that data. It's either using an enhanced HTML, or using the power of JavaScript itself to generate the HTML. Both approaches have advantages and disadvantages. React embraces the latter one, with the argument that the advantages are stronger than the disadvantages.

Why Redux?

Redux is a state management tool for JavaScript applications. While it is frequently used with React, it is compatible with many other React-like frameworks such as Preact and Inferno as well as Angular and even just plain JavaScript. The main concept behind Redux is that the entire state of an application is stored in one central location. Each component of an application can have direct access to the state of the application without having to send props down to child components or using callback functions to send data back up to a parent.

React is great, and it's entirely possible to write a complete application using nothing but React. However, as an application gets more complex, sometimes it's not as

straightforward to use plain old React. Using a state management library like Redux can alleviate some of the issues that crop up in more complex applications.

How is the Client Designed?

Redux architecture revolves around a **strict unidirectional data flow**.
This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand. It also encourages data normalization, so that you don't end up with multiple, independent copies of the same data that are unaware of one another. For the purposes of my application, building upon the principles of Redux, the react-redux architecture was designed as the following.

Action Creators

Actions are payloads of information that send data from your application to your store. They are the *only* source of information for the store. The action creators in this project were all usually related to triggering networking events such as PATCH request to update the data. The minimum required action creators for this project ended up responsible for user authentication and modifying resources such as Areas and Tasks.

Containers and Presentational Components

1. Presentational Components

- Are concerned with *how things look*.
- May contain both presentational and container components inside, and usually have some DOM markup and styles of their own.
- Often allow containment via *this.props.children*.
- Have no dependencies on the rest of the app, such as Flux actions or stores.
- Don't specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it's UI state rather than data).
- Are written as functional components unless they need state, lifecycle hooks, or performance optimizations.
- Since my application is very visual, the presentational components play a huge role in the client. Components include Area, Task, Column, Login, Sidebar and Checkbox. Open source libraries such as react-beautiful-dnd (Drag and Drop) was also added to the application to provide kanban and reorder ability support.

2. Containers

- Are concerned with *how things work*.

- May contain both presentational and container components inside but usually don't have any DOM markup of their own except for some wrapping divs, and never have any styles.
- Provide the data and behavior to presentational or other container components.
- Call Redux actions and provide these as callbacks to the presentational components.
- Are often stateful, as they tend to serve as data sources.
- Are usually generated using higher order components such as *connect()* from React Redux, *createContainer()* from Relay, or *Container.create()* from Flux Utils, rather than written by hand.
- The main container that contains our entire application is AppContainer. The data for the entire application starts at the store and is propagated along the react component tree and changes to the store are reflected automatically.

Benefits of Container-Component Abstraction

- Better separation of concerns. We understand our app and our UI better by writing components this way.
- Better reusability. We can use the same presentational component with completely different state sources, and turn those into separate container components that can be further reused.
- Presentational components are essentially out app's "palette". We can put them on a single page and tweak the design without touching the app's logic. We can run screenshot regression tests on that page.
- This forces us to extract "layout components" such as *Sidebar, Task, Area* and use *this.props.children* instead of duplicating the same markup and layout in several container components.

Reducers

Reducers specify how the application's state changes in response to actions sent to the store. The actions only describe what happened, but don't describe how the application's state changes. Reducers are also pure functions, i.e. they return a new state on every action trigger. This prevents mutations in the existing state. Since the client's store is relatively big, I split the reducers into smaller parts that have control over a small part of the state and combine them to form the global state. Usually actions map to reducers and the client has reducers to update the results for Area, Tasks, Sidebar etc. (src/reducers/*).

API Server - Why Express?

Written in JavaScript, Express acts only as a thin layer of core Web application features. Unlike a big, highly opinionated framework like Ruby on Rails, Express has no out-of-the-box object relational mapping or templating engine. Express isn't built around specific components, having "no opinion" regarding what technologies we plug into it. It strives to put control in the developer's hands and make Web application development for Node.js easier. This freedom, coupled with lightning fast setup and the pure JavaScript environment of Node, makes Express a strong candidate for quick development and rapid prototyping. Express is most popular with startups that want to build a product as quickly as possible and don't have very much legacy code.

The Node.js environment is a key part of what makes Express so easy to build and deploy. Node can be thought of as a cross-platform JavaScript interpreter, able to run JavaScript free from the confines of the browser. In short, Node allows developers to write server-side code for Web apps and networking tools using JavaScript instead of another server-side language like PHP, Python, Java, etc. Using Node Package Manager, or npm, we can install libraries like Express to customize an existing Node installation to our needs. Node users have made thousands of these open-source libraries and server frameworks available.

This is the main library used in our webserver.

Other considerations - Firebase

Firebase is best thought of as a persistent hash datastructure in the cloud. Firebase database can be modeled as elements in a set where we can retrieve or set values via a key. Abstractly, Firebase is like a JSON object. The main benefit of the realtime database is that our data is synchronized across all of your clients. In practice, this means that we have to deviate a bit from the initial conceptional model of our React database as a simple JSON. While setting a property is similar, we don't simply get a value with the property. Instead, we subscribe to a channel for changes to that property. With that in mind, you design a Firebase client differently than you would with a traditional REST API.

Shortcomings

As stated, Firebase's main benefit is the ability to *react* to changes on your collections and elements. Doing this in a fast, scalable realtime way means that Firebase has to impose a few constraints.

If we need deep querying, (example: “Find me all the Tasks with the following Tag from last month”), Firebase lacks this support.

We cannot reverse items in a collection as they come down. (Useful to sort our tasks)

At best, we can apply constraints to one property and order by it. This makes sense from the subscription of changes point of view, but not for most use cases.
One workaround/hack I've seen is to download the whole dataset and reverse it locally. This gets really complicated if your list is very big, and again, is going completely against the grain of Firebase.

Overall, Express gives us infinite control over our server and is not dependent on a third party service like Google’s Firebase which made Express along with MongoDB a better choice.

Database

Relational vs. Non-Relational

Relational databases like MySQL, PostgreSQL and SQLite3 represent and store data in tables and rows. They're based on a branch of algebraic set theory known as relational algebra. Meanwhile, non-relational databases like MongoDB represent data in collections of JSON documents. The Mongo import utility can import JSON, CSV and TSV file formats. Mongo query targets of data are technically represented as BSON (binary JASON). (https://www.pluralsight.com/blog/software-development/relational-non-relational-databases)

Why Non-Relational (NoSQL)?

- **Flexible Data Model**. NoSQL databases emerged to address the requirements for the data we see dominating modern applications. Whether document, graph, key-value, or wide-column, all of them offer a flexible data model, making it easy to store and combine data of any structure and allow dynamic modification of the schema without downtime or performance impact.
- **Scalability and Performance**. NoSQL databases were all built with a focus on scalability, so they all include some form of sharding or partitioning. This allows the database to scale out on commodity hardware deployed on-premises or in the cloud, enabling almost unlimited growth with higher throughput and lower latency than relational databases.
- **Always-On Global Deployments.** NoSQL databases are designed for highly available systems that provide a consistent, high quality experience for users all over the world. They are designed to run across many nodes, including replication to automatically synchronize data across servers, racks, and data centers.

MongoDB is a *document-based* database management system which leverages a JSON-style storage format known as ***binary JSON***, or BSON, to achieve high throughput. BSON makes it easy for applications to extract and manipulate data, as well as allowing properties to be efficiently indexed, mapped, and nested in support of complex query operations and expressions. Mongo is a popular non-relational database for React, Express and Node applications since it is also written in JavaScript and provides god interoperability support.

One other consideration we need to make is whether to use the naive Mongo Driver for NodeJS or use a ODM (Object Data Modeling) library like Mongoose.

Pros/Cons of using Mongoose:

Pros:

- Biggest Pro is that it has the data validation built into it(requirements of what data you will allow to be added or to update your database). Although it requires some work.
- It will abstract away most of the mongoDB code from the rest of the application.

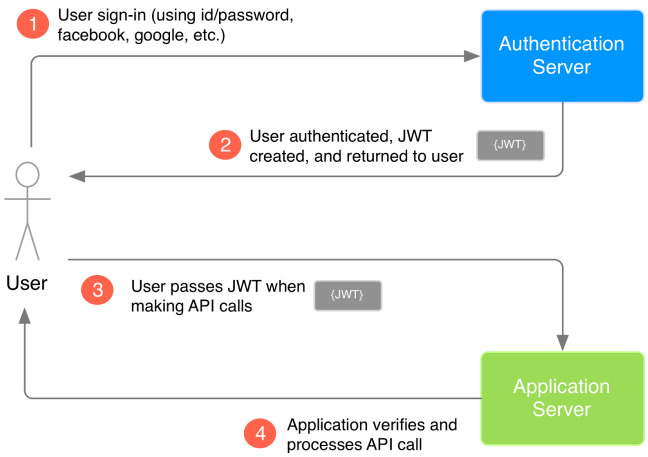
Cons

- Schemas will really defeat the purpose of using NoSQL and it will be hard to experience what is good about having a loose structured data system during the stages of rapid development.
- Not all of our data operations will nicely fit into a characterization that can be encapsulated with a model. Encapsulation is especially hard initially - unless we have a very clear idea of the data flow before you start (which is ideal, but not easy when you are building something conceptually new and requires a lot of experimentation and change/redesign).

User Authentication

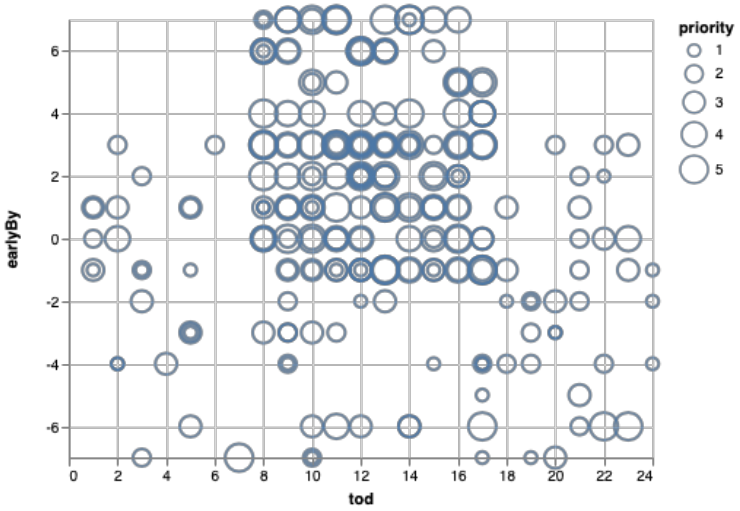
A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties. The token is composed of a header, a payload, and a signature. In order to save the user in the database and later compare the password he/she enters for generating the authentication token, we need to encrypt his/her password, as it's not safe to keep passwords with no encryption in the database. Also, it would be a joke to use md5 for that and sha1 recently became unsafe.

From Wikipedia: "bcrypt is a password hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher, and presented at USENIX in 1999". Here is more on why you should use bcrypt to hash passwords.



Exploratory Data Analysis and Transformation

I used Pandas, Numpy and Matplotlib to visualize the data for our test user's data. I used 2 main feautures, Time of Day (tod) and priority to predict how early a user might complete a task (earlyBy). The data I used was extracted from my Google calendar and missing fields were manually entered. The calendar data file had unnecessary data so I transformed the csv file into only the features we require using a simple python script. Here is a visualization using Altair that I used to look for patterns. It is evident that in my case, most of my work was between 8am and 5pm. The scale for tod used in the chart is a 24 hour clock.



Model Creation

In a *regression* problem, we aim to predict the output of a continuous value, like a price or a probability. Contrast this with a *classification* problem, where we aim to predict a discrete label (for example, where a picture contains an apple or an orange).

Since in our case we want to predict a continuous value such as earlyBy which is clearly a regression problem.

Considerations to choose an algorithm

Accuracy

Getting the most accurate answer possible isn't always necessary. Sometimes an approximation is adequate, depending on what you want to use it for. If that's the case, you may be able to cut your processing time dramatically by sticking with more approximate methods. Another advantage of more approximate methods is that they naturally tend to avoid overfitting.

Training time

The number of minutes or hours necessary to train a model varies a great deal between algorithms. Training time is often closely tied to accuracy—one typically accompanies the other. In addition, some algorithms are more sensitive to the number of data points than others. When time is limited it can drive the choice of algorithm, especially when the data set is large.

Linearity

Lots of machine learning algorithms make use of linearity. Linear classification algorithms assume that classes can be separated by a straight line (or its higher-dimensional analog). These include logistic regression and support vector machines (as implemented in Azure Machine Learning). Linear regression algorithms assume that data trends follow a straight line. These assumptions aren't bad for some problems, but on others they bring accuracy down.

Building the Model

Let's build our model. Here, we'll use a `Sequential` model with two densely connected hidden layers, and an output layer that returns a single, continuous value. The model building steps are wrapped in a function, `build_model`, since we'll create a second model, later on. The code that builds my model is presented in the next page.

The loss function and evauation metrics chosen are as follows:

- Mean Squared Error (MSE) is a common loss function used for regression problems (different than classification problems).
- Similarly, evaluation metrics used for regression differ from classification. A common regression metric is Mean Absolute Error (MAE).

```
from keras import optimizers

def build_model():
  model = keras.Sequential([
    keras.layers.Dense(5, activation=tf.nn.relu,
                       input_shape=(train_data.shape[1],)),
    keras.layers.Dense(16, activation=tf.nn.relu),
    keras.layers.Dense(1)
  ])

  optimizer = tf.train.RMSPropOptimizer(0.001)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae'])
  return model

checkpoint_path = "training_1/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create checkpoint callback
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)

model = build_model()
model.save('my_model.h5')
model.summary()
```

Training the Model

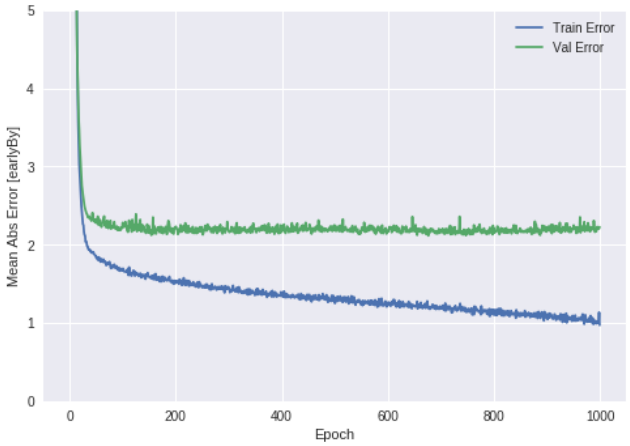
The model is trained for 500 epochs, and record the training and validation accuracy in the `history` object.

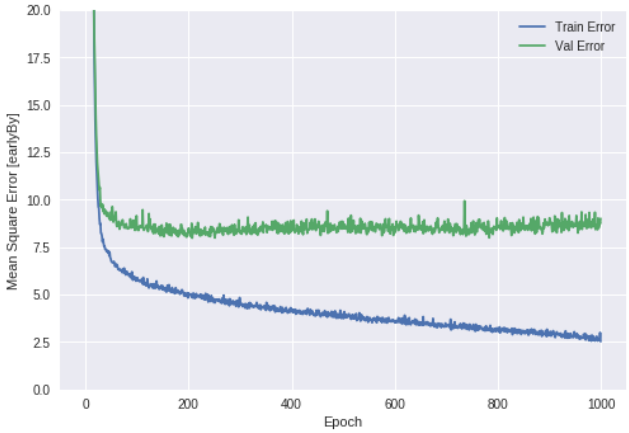
```
class PrintDot(keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs):
    if epoch % 100 == 0: print('')
    print('.', end='')

EPOCHS = 500

# Store training stats
history = model.fit(train_data, train_labels, epochs=EPOCHS,
                    validation_split=0.2, verbose=0,
                    callbacks=[PrintDot()])
```

This graph shows little improvement, or even degradation in the validation error after a few hundred epochs. Let's update the `model.fit` method to automatically stop training when the validation score doesn't improve. We'll use a *callback* that tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then automatically stop the training.

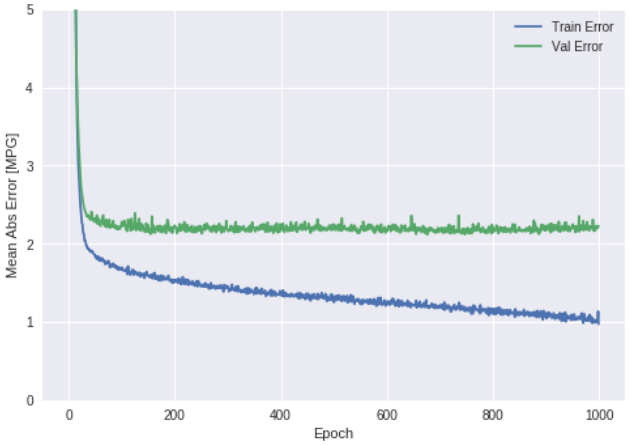


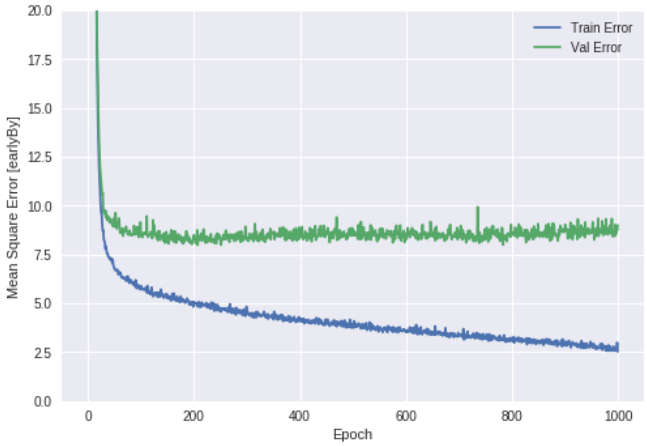



```
model = build_model()

# The patience parameter is the amount of epochs to check for improvement
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=50)

history = model.fit(normed_train_data, train_labels, epochs=EPOCHS,
                                                    validation_split =
```



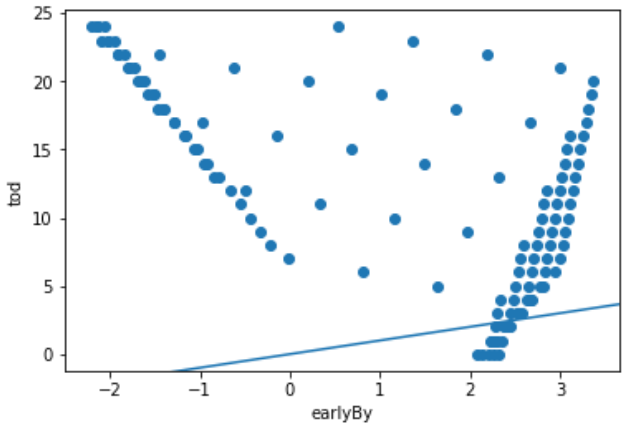



```
0.2, verbose=0,
callbacks=[early_stop, PrintDot()])
plot_history(history)
```

The graph shows that on the validation set, the average error usually aeound +/- 2 MPG. The testing set mean abs error came out to 1.85 and prediction accuracy of 76%.

Visualizing the model

To visualize this model I programmatically generated random tasks to see how the model would predict how early a task is. The model predicts that a task will be completed early if the time of day is between 5am to 8pm. Also the model predicts that



tasks are usually completed later as we get close to midnight.

Machine Learning Conclusion:

- When input data features have values with different ranges, each feature should be scaled independently.
- If there is not much training data, prefer a small network with few hidden layers to avoid overfitting.
- Early stopping is a useful technique to prevent overfitting.

Conclusion and Future

With a modest prediction accuracy and lots of data it is feasible to predict productivity at a given time for a user. Since this data is not already available these will not be available in the application until the user completes enough tasks on our platform. Another way to make this better in the future is to cluster similar users using an unsupervised clustering algorithm such as K-means and perform model training on data from each cluster since similar users will have similar productivity.

I will be continuing to work on this application for the next year and launch it along with mobile application on the iOS and Android along with the already built out backend and web interface. One of the most important considerations for this project was to ship it at the end. The application is currently hosted at oedo.netlify.io.

References and Resources

1. https://redux.js.org - A predictable state container for JavaScript apps.
2. https://reactjs.org - A JavaScript library for building user interfaces
3. https://www.sketchapp.com - Visual Design Prototyping tool
4. https://expressjs.com - Fast, unopinionated, minimalist web framework for Node.js
5. https://www.mongodb.com - Non-relational database
6. https://mongoosejs.com - elegant mongodb object modeling for node.js
7. https://jwt.io - JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.
8. https://jakevdp.github.io/PythonDataScienceHandbook/

All source code available at gitlab.com/oedo