12-1-2017

A Test Driven Approach to Develop Web-Based Machine Learning Applications

Armin Esmaeilzadeh
University of Nevada, Las Vegas, armin.esmaeilzadeh@gmail.com

A TEST DRIVEN APPROACH TO DEVELOP WEB-BASED MACHINE LEARNING APPLICATIONS

by

Armin EsmaeilZadeh

Bachelor of Science (B.Sc.)
Azad University, Iran
2014

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2017

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

October 3, 2017

This thesis prepared by

Armin EsmaeilZadeh

entitled

A Test Driven Approach to Develop Web-Based Machine Learning Applications

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Kazem Taghva, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Ajoy Datta, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Abstract

The purpose of this thesis is to propose the design and architecture of a testable, scalable, and efficient web-based application that models and implements machine learning applications in cancer prediction. There are various components that form the architecture of our web-based application including server, database, programming language, web framework, and front-end design. There are also other factors associated with our application such as testability, scalability, performance, and design pattern. Our main focus in this thesis is on the testability of the system while considering the importance of other factors as well.

The data set for our application is a subset of the Surveillance, Epidemiology, and End Results (`SEER`) Program of the National Cancer Institute. The application is implemented with Python as the back-end programming language, Django as the web framework, Sqlite as the database, and the built-in server of the Django framework. The front layer of the application is built using `HTML`, `CSS` and various `JavaScript` libraries.

Our Implementation and Installation is augmented with testing phase that include `unit` and `functional` testing. There are other layers such as deploying, caching, security, and scaling that will be briefly discussed.

Acknowledgements

"Foremost, I would like to sincerely thank my thesis adviser Dr. Kazem Taghva for his support of my M.Sc. study and research and giving me the opportunity to work with him. His advice, motivation and immense knowledge has given me guidance in my research and writing this thesis.

Furthermore, I would like to thank my committee members, Dr. Ajoy K. Datta, Dr. Laxmi Gewali, and Dr. Emma Regentova for their support and for being part of my thesis committee.

And above all, I would like to express my profound gratitude to my parents Homayoun Esmaeilzadeh and Fariba Heidari, my sister Arina Esmaeilzadeh and my cousin Dara Nyknahad for providing me with continuous encouragement and support throughout my years of study. My accomplishments would not be possible without them."

Armin EsmaeilZadeh

University of Nevada, Las Vegas

December 2017

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The objective of this thesis is to propose the design and implementation of an interactive web based application that can be used to host Machine Learning models. Our main software/application development is based on test-driven development. In the first cycle of this approach, we define a specific set of test cases based on the requirements of the application. In the second cycle, we implement the corresponding routines to functionally pass the test cases at any level of the application such as database, front-end, service, controller, etc that are required. Once we verified that all the test cases are passed then we repeat the first 2 cycles again for any new requirement or refactoring. This approach will allow us to be certain that every new functionality that is added to the application meets the requirement in the first cycle of the development.

The Machine Learning technique that is used in this thesis is a Logistic Regression model for predicting the 5-years survivability for breast cancer incidents. The data set that was used to train the model is a part of the Surveillance, Epidemiology, and End Results (`SEER`) program of the National Cancer Institute (NCI).

1.1 Outline

In chapter 2, we give a brief introduction to Machine Learning. We will introduce three of the most common used algorithms in the field of Machine Learning which are Decision Trees, Linear Regression, and Kmean Clustering. We also give a description of the Model that is used in this thesis.

Chapter 3 will give a detailed description of the Django framework that will be used to implement our web application. We will look into details of some of the most important aspect of the framework such as the server, the structure of the framework, and the Model View Template (`MVT`) architecture.

In chapter 4, we will introduce the testing framework of the Django framework. We will define the structure of the tests and how to run test cases. The `MVT` architecture of the Django framework will be discussed with many examples of successful test cases.

Chapter 5 will describe the implementation phase of the project. We will present the implementation of the important components of the project and three different applications that are

built.

In chapter 6, we will present our web application that includes different components of the Django framework to construct the application.

Chapter 2

Machine Learning

2.1 Introduction

Machine Learning is a field of the Computer Science that gives computers the ability to learn from experience without being explicitly programmed [Sam59]. More specifically, the field of machine learning explores and studies the design of algorithms that can learn from data and make predictions on new and unseen examples. This will greatly overcome the problem of program instructions that are defined strictly by the programmer and will allow the computers to make data-driven decisions and predictions dynamically.

Machine learning has evolved from the study of pattern recognition in the field of Artificial Intelligence during 1950s and gained more popularity during recent decade as we had experienced a rapid increase in the amount of published data and a dramatic growth in performance and speed of CPU and GPU in processing data and information. There are also many fields in Computer Science and Mathematics that are very closely related to Machine Learning such as statistical learning, mathematical optimization, and data mining. In fact, many ideas in the field of Machine Learning from practical methodologies to theoretical concepts had a long historically in mathematics.

In this chapter, we will look at different types of Machine Learning algorithms and their applications.

2.2 Categories

Typically, machine learning algorithms are classified into three general categories depending on the nature of the data and learning methodologies used in the learning system. The three categories are [Rus03]:

- **Supervised learning**: Models in this category are given a set of manually selected examples and features as inputs and the goal of the algorithm is to learn the rules and principles contained in the data.
- **Unsupervised learning**: The algorithms are not given examples as input. It is left to the model or algorithm to find relationships of patterns in the input data.
- **Reinforcement learning**: The algorithm or the program will interact with environment and it has to perform a certain task or goal. During this period, the algorithm is getting feed backs regarding its performance as it explores the problem space.

We can also categorize machine learning algorithms based on the type of predictions and the desired output that we expect the machine learning system to provide. Here are some of the most important categories [Rus03]:

- **Classification**: The input data that is given to the algorithm is divided into two or more different classes. The goal of the system is to find principles related to each one of those classes and be able to assign a class value to a new and unseen input. Most of the algorithms in this branch are supervised learning such as classifying emails as spam or non-spam.
- **Regression**: This method is also very similar to classification except that the outputs of the system are continuous values rather than discrete.
- **Clustering**: The goal of these algorithms is to divide the given input data into different groups but unlike classification methods, the groups and classes are not known in advance. Therefore putting this models in unsupervised category.

In the following section, we give an example model in the categories of Classification, Regression, and Clustering to have a better understanding of how these models work and their use cases.

2.3 Machine Learning Models

2.3.1 Decision Tree

Decision Tree is most commonly used in Classification problems and it is a supervised learning model [Gai]. It uses a tree like model or graph to represent different decisions that can be made at each level of the tree, based on features that have the most predictive power at that level and the consequences of that decision. Therefore, each internal node of the tree is representing an attribute of the data and each branch or edge is an outcome of the related node. Finally, each leaf node at the lowest level of the tree will give us the class or label of the dependent variable. The path that was taken from the root of the tree to the final leaf is the classification rule.

The algorithms that are used in Decision Trees work in a top down manner. It chooses an attribute in the data, that best splits the data or set of items, to be at the top of the tree or its root. Based on the value of that attribute it will go to the next level of the tree and chooses two other attribute that split the data set. this process will go on until we reach a leaf node. There are different metrics that are used to find the best attribute at each level of the tree such as Gini impurity [Bre84a], Information gain [Mit97] or Variance reduction[Bre84b]. The tree, by using these metrics, will be able to learn and find the best attributes by splitting the data sets into subsets many times. This process is called recursive partitioning. The recursive process will stop if the splitting procedure is no longer adding any value to the prediction. This process of top down induction is an example of greedy algorithms that is so far the best strategy to learn classification rules in the data.

One of the well-known methods that is used to split the data set into subsets that are homogeneous (subsets that have instances with has similar values) is Entropy[Mit97]. The Entropy calculates the homogeneity of the data in the subsets. If the data set is totally homogeneous then the entropy would be zero. If the data set is equally divided the entropy would be one. Now in order to build the decision tree we must calculate two types of Entropy:

- Entropy by using the frequency table of one feature [Gai]:

$$E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2.1}$$

- Entropy by using the frequency table of two features [Gai]:

$$E(T, X) = \sum_{c \in X} P(c)E(C) \tag{2.2}$$

Now, once we have these two formulas we can start finding the best attribute that can be used at the root of the tree. First, we have to find the Entropy of the target variable, and then we calculate the Entropy of each one of the attributes with the target variable. The attribute that gives us the largest value is therefore providing the best Information Gain. We use that variable at the root of the tree and we move on to find the next nodes. The recursive function will continue until all of our data is classified.

2.3.2 Linear Regression

Linear regression models attempt to find a linear equation between one or more feature variables and a target variable based on the observed data [Sea67]. As example, a linear regression model may relate the weight of individuals to their height using a linear equation. But before any attempt to find such an equation we must first make sure that there exist a relationship between these variables. The relationship does not necessarily imply causation but rather a significant association between variables. A widely used data visualization tool that is used to observe if there exist any relationship between variables is scatter plot. If the supposed relationship does not exist in between the explanatory and dependent variables then a linear regression model will not fit the data and it will not be a useful method.

One of the measures that is used to determine the association between variables is correlation coefficient which will give us a value between -1 and 1 to indicate the strength of the relationship between two variables in the observed data. The Pearson correlation coefficient formula is [oC]:

$$\rho(X,Y) = \frac{\mathbf{Cov}(X,Y)}{\sqrt{\mathbf{Var}(X)\mathbf{Var}(Y)}}. \tag{2.3}$$

The most common method that is used to fit a regression line on the observed data is least-squares method. This method will find the best line that fits the observed data by calculating and minimizing the sum of the squares of the deviations of each data point from the line. The general formula for the sum of the squared residuals (residual is the difference between the observed value and the value provided by the model) is [Dra98]:

$$SSE = \sum_{j=1}^{n} r_j^2 = \sum_{j=1}^{n} (y_j - \hat{y}_j)^2 \tag{2.4}$$

Eventually the final form of the equation of the fitted line in the linear regression model is:

$$Y = a + bX \tag{2.5}$$

Where X represents the explanatory variable and the Y represents the dependent variable. The equation has a slop of b and the intercept of the line is a. By having this equation, if a new unseen

value of X is presented without the accompanying Y value, we can replace the X variable in the equation with given value and predict the value of Y.

2.3.3 Kmean Clustering

Kmean Clustering is method originated in the field of signal processing with applications in Vector quantization, Image segmentation, etc. This method is popular and widely used in cluster analysis. The goal of this method is to partition n observations or given data points into k clusters where each data point belongs to the cluster with the closest mean. Computationally, this problem is an NP-hard [Mah09]. But there are different heuristic algorithms that can be used to converge to a local optimum [PT12]. The given data to this method has a set of features but it has no labels or target variables that we want to predict. Therefore, Kmean clustering method is an unsupervised learning algorithm.

A common algorithm used in Kmean clustering uses an iterative refinement techniques and is also referred to as Lloyd's algorithm [Bho09]. In the Kmeam algorithm we are getting a set of training examples $x_1, x_2, ..., x_m \in \mathbb{R}^n$ and we want to group these observations into a few clusters. The goal is to find k centroids and assign a label $c^{(i)}$ to each one of the data points determining its cluster. There are mainly two steps involved in these method [PT12]:

- Initialization:
  - In this step we initialize the cluster centroids $\mu_1, \mu_2, ..., \mu_k \in \mathbb{R}^n$ randomly.
- Repeat until Convergence:
  - Assigning step: for every data point, find its closest cluster using the squared Euclidean distance and assign the data point to that cluster [PT12].

$$c^i := \arg\min_j ||x^i - \mu_j||^2 \tag{2.6}$$

  - Update Step: for every k cluster centroid calculate new means or centroid based on the new observations that was added to those clusters [PT12].

$$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}} \tag{2.7}$$

The algorithm will converge when the assignments are not changing anymore.

2.4 Machine Learning in Cancer Research

Machine Learning models such as decision trees and artificial neural networks have been used in cancer research for over 20 years and it is not a new topic [Wis07]. However, the majority of these applications have been used to detect, classify and distinguish malignancies which acted as an aid in cancer detection and diagnosis. But through the last decade we have seen a growing trend in applying machine learning models in cancer prognosis and prediction as well. The objectives of cancer prediction is mainly in three areas [Wis07]:

- Predicting Cancer Susceptibility.
- Predicting Cancer Recurrence.
- Predicting Cancer Serviceability.

In the first case, we are trying to predict the development of cancer type prior to its occurrence. In the second case, we will try to predict the recurrence and redevelopment of a type of cancer after a resolution for it. In the third field, the objective is to predict survivability, life expectancy or any other specific outcome after the diagnosis.

Nevertheless, cancer prognosis need to consider and take into account more information and data about patient and the disease than just the diagnosis. Historically the process of prognosis involved many physicians with different specialties to look at different clinical factors, information about tumor and other data about the general health and life style of patients to find a sensible prognosis. However, with the vast development of different technologies in imaging (fMRI, PET), genomic (DNA, microarrays) and proteomic (protein chips, tissue arrays) fields, the scale of molecular information about tumors have expanded tremendously. Aside from these data we have other biomarkers, clinical factors, demographic, histological and macro level information about patients that no traditional approach in cancer prognosis and prediction by physicians is practical. Moreover, it has been shown that these molecular level data have very powerful prognostic and predictive attribute [Wis07].

Therefore, we have seen a growing trend in applying computer based models such as Machine Learning to tackle the intensive computational challenge of processing and analyzing these data and information. Moreover, the use of these techniques is helping us to move toward personalized medicine which is important not only for patients but also for physicians to make better decision and for the general economy and policy makers.

2.4.1 5-year Surivavability Prediction of Breast Cancer

Surveillance, Epidemiology, and End Results (`SEER`) program of the National Cancer Institute (NCI) has been collecting and publishing cancer data from approximately %28 of the population in the United States of America since 1970s [Sp]. The data set includes demographics, clinical data and information regarding tumor size, stage, etc of patients. `SEER` data set is the only comprehensive source of information for cancer diagnosis and patient survival data in the United States. The machine learning model that we have used in the web application in this thesis is a Logistic Regression model trained on the breast cancer data available in `SEER` program data set and it is trained to predict the 5-year survivability rate for breast cancer observations.

There are the following 15 attributes in the data set that have the most predictive power in survivability prediction. The Logistic Regression model have determined the coefficient for each one of these attribute and they will be used to predict the survivability rate for any new and unseen observations.

Variable	Variable Definition
race	two-digit code race identifier
maritalStatus	one-digit code for marital status
behaviorCode	code for benign etc.
grade	cancer grade
vitalStatusRecord	alive or not
histologicType	microscopic composition of cells
csExtension	extension of tumor
csLymphNode	involvement of lymph nodes
radiation	radiation type code
SEERHistoricStageA	codes for stages
ageAtDiagnosis	First diagnosis age
csTumorSize	size in millimeters
regionalNodesPositive	negative vs positive nodes
regionalNodesExamined	positive and negative nodes examined
survivalMonths	number of months alive

Table 2.1: 15 Attributes of Logistic Regression Model.

Chapter 3

Django and Web Developing

3.1 Introduction

Django is an open source and free web framework [FC] that supports developing testable, scalable and secure web applications [Sto] [tBPV] [Dja] built in Python. It was built by Adrian Holovaty and Simon Willison at the Laweence Journal-World newspaper in 2003 and was publicly released under BSD licence in 2005. Django Software Foundation currently maintains the Django project.

Django provides a set of components and rich libraries for database access, template processing, session management, user authentication that are needed to develop web applications. It follows the Model-View-Template(`MVT`) architectural pattern that is extensively used to build modern web applications and web services [Phi].

In the following sections, we will review different components and technologies that are available in the Django framework. We also discuss various components that form the architecture of the web application and how they interact with each other.

3.2 Server

One of the fundamental functions of any web application is to receive requests and return responses. Like many other web applications, request and responses in Django have two phases, namely `Request` Phase and `Response` Phase. We will look at these two phases in more detail in the following sections:

3.2.1 Request Phase

In a Django based web application we have to configure the web server to route incoming requests to a script that has implemented `WSGI`(Web Server Gateway Interface). The WSGI's specification is written to map the incoming `HTTP` requests into Python objects and then execute one or more applications in a stack [WSG]. The WSGI application at the bottom of the stack is the Django application. Those applications that reside on the top of the Django application are called `middle-ware`. The middle-wares are used to preprocess the incoming requests. For low level operations such as logging we can make use of these middle-wares. The first stage in the request phase happens in the web server. Here We should note that for the purpose of our example, we will not be using any separate web server, the web application will run locally and any request call will be directed to the WSGI server in the web application. Regardless of which web server we chose, its primary function is to accept the requests coming from clients and decide where the request should be directed to. Most of the web servers would pass the requests to files, scripts, or other defined processes.

Once the Django application receives the request, it will wrap it in an HTTP request object that is defined in the Django application. From this point on, any part of the application that deals with the requests will use this HTTP Request object. Before processing requests, Django emits a `request_started` signal in the application. This signal is subscribable by other applications that may want to do some operations, such as clearing the logs of data base queries that has been executed before. Here, there are middle-wares that are specific to the Django framework. After emitting the signal, Django will pass the request object to the stack of middle-wares. Any Django middle-ware in the Django framework is a class that is implementing one of the following functions:

- `process_request`
- `process_view`
- `process_template_response`
- `process_response`
- `process_exception`

These middle-wares are arranged in a stack in different orders. Django will call middle-ware functions at different times and in different orders. For example, in the request phase the methods are called in ascending order and in the response phase they are called in descending order. During the request phase, Django calls the `process_request` method in any middle-ware class that implemented that method, in ascending order. Usually the process_request method is used in two ways. It is either added to the HTTP request object or to a preemptively return response. For instance, Django's session middle-ware will add a session attribute to the request and the authentication middle-ware adds a user attribute to the request. In case the request should be denied based on its header, Django's common middle-ware will preemptively return a response. In this case, no other process_request method in the middle-wares are being executed and the returned response will be passed to the process_response methods which will be discussed later.

After the Django is finished calling the process_request methods in the middle-ware stack, it will load its `URL` configuration pattern. The default configuration for the URLs are defined in the project settings in the `urls.py`. However, if any of the middle-ware classes add a urlconf='...' attribute to the request object, Django will use that instead of the default settings. Any URL configuration is a hierarchical list of regular expressions that are used to route a requested path inside the request object to a view in the application.

Once the Django matches the URLs path to a view, the object will be sent to the middle-ware stack to call the `process_view` method in middle-ware classes that has implemented it. This step will allow the middle-ware classes to inspect the view function that Django is going to execute and the arguments it will pass to it. The middle-ware can alter the arguments or return their own responses. For example, Django's Cross Site Request Forgery (`CSRF`) middle-ware uses this method

to see if the view function has an attribute that marks it as exempt from checking.

The final step in the request phase is the view. The view is a callable Python object that takes an HTTP request object and returns an HTTP response object. Also, the views are able to accept arguments that are parts of the request path.

3.2.2 Response Phase

The logic of our web application is inside the view functions (which act as callable objects). These view functions are different from presentation logic which is handled by the template. In simple terms, the business logic in the view decides what data can be shown based on the processing that has been done on the data by the logic. The presentation logic will decide if and how the data is going to be presented. Hence, the view is responsible to collect all the data that is required to form the response. The final data will be put inside a context dictionary. After this point the view can perform one of the two options. It can render the template itself and return it inside an HTTP response or it uses a template response. In the latter case, Django will call the process_template_response method in the middle-ware classes in the stack that have implemented it in reverse order. These middle-wares are able to alter the template or the context data.

Regardless of whether the view has rendered the template or Django has called the response render function, the template will be rendered. In order to render a template, a context dictionary that has the data and a template is needed. Any variable that is defined inside the template will be substituted with the values associated with that key in the context dictionary. Templates also have access to filters that are used to provide special formatting to the data.

At the final stage, Django will go through the middle-ware stack in reverse order to call process_response methods in each of the middle-ware classes that have implemented it. The process_response method must return an HTTP response object even if it is the same object that was passed to it. It can make its own response or alter the existing one.

After Django is finished with middle-ware stack, it will emit a request_finished signal which similar to the request_started signals in the beginning of the process is subscribable by other appli-

cations. Applications can use this opportunity to handle any clean up tasks.

Django will then convert its own HTTP response object to an appropriate WSGI response which will be sent back to the WSGI middle-ware stack for further processing. Finally, the WSGI response is returned to the web server that sent the request. The web server will then convert the WSGI response into its own format and will direct it back to the user.

3.3 MVC and MVT Patterns

The `MVC` stands for Model, View and Controller. It is a design pattern or software development methodology that is being used to develop web applications. The main objective of this design pattern is to promote code reusability. For this reason, the MVC pattern divides the software or web application into three main components namely Model, View, and Controller. The responsibility of each component is described below:

- **Model**: This component contains the business logic of the applications. All the rules, tasks and processes that will be applied to the data is defined here. It is an interface to the data so it can retrieve data from the database without knowing the details of underlying database. So, the same model can be used to interact with different databases.
- **View**: This is the presentation layer. It has the information on how to present the data on the display such as the layouts, colors, etc. The view components will be the user interface so it can be used to collect user inputs as well.
- **Controller**: This component will handle the communication between incoming requests from users and the models. So basically it will control the flow of information between view and the model. It will also get the information that has been collected from user by the view and will decide to either change the view or modify the data through the model.

So, the general flow will be as follow: The user will interact with the interface that has been produced by the view. The interaction will be passed to the controller, and the controller will decide if any request should be send to any model. The models will receive the request, will do the data processing that is required and then it will pass the data back to the controller. The controller then uses the data received from the model and will pass it to the view.

While this is the general approach of the MVC pattern, different frameworks have different interpretation of how to implement it. Django is using the MVC pattern very closely, however it uses some of its own logic for implementation. The main difference is that the controller component is being handled by the framework. So, any request goes directly to the view classes, which have the logic of the application and different methods to handle requests based on the request paths. The models are object entities that reflect the tables that are in the database. The view will use this model objects to retrieve, update, delete or do any other processing task on the data. The

view then will use templates, which are the presentation layer, to substitute data with variables in these templates. For these reasons the pattern that Django uses is named `MVT` which stands for Model, View, and Template. The definition of each component is described below:

- **Model** is the data access layer. It is an object that mirrors the tables that are in the database. It includes the fields and methods to do some data processing on the data such as validations.
- **Template** is the presentation layer. This component acts as the view component of the original MVC pattern. It has all the information about how to present the data such as HTML layouts and `CSS` and `JavaScript` codes.
- **View** is the business logic layer. It has the rules to access the data through models and pass them to the appropriate templates.

As we saw the differences are mainly in naming the components. The underlying pattern is the same. So, anybody who has used the MVC pattern must be comfortable to work with the MVT pattern of Django.

3.4 Environment

In software architecture and software deployment, a tier or environment is a system in which the program or application is deployed and executed. Deployment architectures vary significantly across industries. But a very common 4-tier architecture is usually development, testing, staging and production tiers which the application is being deployed to each one in order. These environments or tiers may vary in size significantly. For example, the development environment is typically a developer's workstation, while the production environment might be a large network of different geographically distributed systems. In the project of this thesis we will look at the development environment and since we will not have the deployment phase we will run the tests in the same development environment.

Once we decided to start and develop the application there are a few general best practices that we can follow to choose a development and production environment. A few of these guidelines are:

- **Isolation**: since in most cases, web applications have multiple dependencies on other applications, middle-wares or tools, it is important to avoid using tools or packages that are installed outside of the development environment as much as possible. It is especially true in case of python packages and libraries that are used for machine learning and data science tasks which are using some C extensions. If these tools or packages are installed at the system level and we have used them inadvertently, we may find that the application is not working properly or as we expected in the production environment or once we share the application to be run on different systems.

- **Determinism**: it simply means that we are confident about the versions of the libraries, packages and tools that our application relies on and we can reproduce that environment reliably.

- **Similarity**: As the complexity and size of the applications increases it is important to be confident that the problems that arise in the production environment can be reproduced in the development environment. This will substantially reduce the scope of investigation and debugging time. Therefore, it is preferable to run the application on the same operating system, the same release and that the same tools have been used to configure the development and production environment.

In Django projects, there are tools that can help us to follow the above guidelines. For example, we will be using a tool called

tt virtualenv to create virtual environments in our systems so that the project is not dependent on the system's site-packages. Also for dependency management we will use pip, which is a python package manager, to install and to specify versions of the packages that need to be installed and used.

The specification of the system and libraries that we used to develop our application is listed below:

Operating System	Linux Ubuntu 16.04.3 LTS
Ram	16 GB
CPU	2.7 GHz
Python	3.6.2
Django Framework	1.11

Table 3.1: System Requirements.

3.5 Installation

3.5.1 Setting up Environment

The development environment of Django consists of installing Python, (any python release after 2.7 already includes pip as package manager and virtualenv to create virtual environments), a database system and a web server. Django makes it easy for developer to focus on developing the applications in the early stages of development by providing a lightweight SQLite 3 database system and a web server.

3.5.2 Installing Python

The following link provides the latest versions of Python and installation instructions:

```
https://www.python.org/downloads/
```

Listing 3.1: Python versions.

3.5.3 Create a Workspace with Virtual Environment

The name of the virtual envirnpment is `myVirtualEnv` and the virtual environment files will be installed inside it.

```
mkdir myVirtualEnv #Create a directory called myVirtualEnv virtualenv
./myVirtualEnv #Install the virtual environment inside the directory
source ./myVirtualEnv/bin/activate #Activate the environment
```

Listing 3.2: Install and activate Virtual Environment.

Once the environment is activated, the name of the environment will be attached to the command line in parenthesis.

3.5.4 Start a requirements file

We can create a `requirements.txt` and write all the packages and dependencies that this program needs. At this point we will only have Django as the main dependency, So we write it inside the text file.

```
Django==1.11
```

Listing 3.3: Django dependency.

3.5.5 Installing Requirements

Since we have the required packages inside the `requirements.txt` we can use python's packages manager, `pip`, to install all of them using the following command.

```
pip install -U -r requirements.txt
```

Listing 3.4: Install dependencies.

3.5.6 Database setup

Django framework supports most of the major database engines to be setup and used. The following engines are some of the `SQL` database systems supported in Django:

- MySQL
- PostgreSQL
- SQLite 3
- Oracle

Django also supports `NoSQL` systems such as search databases, in memory data structures, document databases and catching systems as well. Some of them are listed below:

- Mango DB
- Google App Engine Data store
- Elastic Search
- Cassandra
- Simple DB
- Redis

As mentioned before, Django comes with the SQLite 3 database and we will be using it throughout this project.

3.5.7 Web Server

As shown in the previous sections, Django comes with its internal lightweight web server that is used for developing and testing applications. The server is pre-configured to work with the framework and more importantly it restarts itself whenever a part of the code is modified, which is very helpful in the early stages of development.

However, Django supports most of the popular web servers such as Apache, Nginx, Cherokee and Lighttpd to name a few. In this project, we will be using the development server that comes with Django.

3.6 Django Project

Some MVC web frameworks use a technique called Scaffolding in which the developer can define some specifications for how the application's database maybe used. The framework then uses this instructions to generate code and project structures to be used as a starting point for developer to start developing the application. Django provides HTTP as well as file system scaffolding.

The `HTTP scaffolding` is used to handle tasks such as parsing an HTTP request and turn them into python objects or providing tools to easily create HTTP responses. The file system scaffolding is a set of conventions for how to organize the code. These structure and organizations makes it easier for developers to add more engineers to the project since most engineers working with Django already know this organization of the code. In Django framework a Project is the final product and it groups one or more applications together.

3.6.1 Creating the Project

Once Django is installed, it will use a `django-admin.py` script to handle scaffolding tasks. Since we are in the virtual environment `myVirtualEnv` we can use `django-admin.py` to start a project by using the following command.

```
1 python django-admin.py startproject myproject .
```

Listing 3.5: Start Project Command

Once the project setup is finished the framework will generate the following files that are the starting points to start development.

- **manage.py**: this script is pointer back to the `django-admin.py` script that was used to start the project. From this point we will be using this script to interact with the framework. `manage.py` has an environment variable set that points to the project to read settings from and operate on.
- **settings.py**: All the configuration of the project will be in this file. It has some default values for different components of the application such as database, web server, etc to use and to develop the application. If we decide to change the database, web server, etc, we first apply the changes here.

- **urls.py**: this file contains the mapping between URLs, as part of the incoming requests, to views in the application.
- **wsgi.py**: this script is a `WSGI` wrapper for the application. The script will be used by the development server of Django and other containers such as mod-wsgi that is used in the Apache web server in the production environment to interact with the project and applications grouped in it.

3.6.2 Creating the Application

At this point we have everything ready to start the application. We will use the following command to start an application:

```
1 python manage.py startapp myapp
```

Listing 3.6: Start Application Command

Once the applications are setup the framework will generate the following files in the `myapp` folder which holds the application:

- **models.py**: This file will contain the Django ORM models for the application.
- **view.py**: This will contain the view code and the logic of the application.
- **test.py**: All the unit and integration tests will be defined here.

3.7 Settings file

All the configurations of the Django project are inside the `settings.py` file. The settings.py file is a python module that has module-level variables. Because the setting file is a module, there should be no Python syntax errors in the file. We can use normal Python syntax to assign values to variables and we can also import values from other settings files.

For example, if we want to have a list of characters we can use the following python syntax:

```
MY_SETTING = [str(i) for i in range(30)]
```

Listing 3.7: Python syntax.

If we want to change the settings file's path or create other setting files, we have to specify which settings file we want to use by assigning a path to the `DJANGO_SETTINGS_MODULE` variable. The value of this variable should be Python path syntax for example: `mysite.settings`. We can export the settings file path to the environment variable using the following command:

```
export DJANGO_SETTINGS_MODULE=mysite.settings
```

Listing 3.8: Django setting module.

Or we can pass the path every time we want to run the server:

```
django-admin runserver --setting=mysite.settings
```

Listing 3.9: Django setting path.

On the live server side, we also have to assign the settings file path to a server environment variable of the WSGI application. It will be done in `wsgi.py` file using `os.environ`:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

Listing 3.10: Django setting path variable.

There is a default settings file in the `django/conf/global_settings.py` that Django uses in case there are no setting files for the application. So generally, Django first loads the `global_settings.py`

and then it loads settings from the specified settings files and overrides the global settings as necessary. We can use the `"python manage.py diffsettings"` command to see the difference between the default settings and the ones that we are overriding.

In case we want to use some of the settings configurations in our applications, we can import the setting files:

```
from django.conf import settings

if settings.DEBUG:
    # Run Something
```

Listing 3.11: Import settings

Here we should note that the settings is not a module, it is an object, so we are not able to import specific variables from the object. Once we have imported the module we have all the configurations.

```
from django.conf.settings import DEBUG # We can't import values.
```

Listing 3.12: Wrong changes in the settings.

However, we should be careful not to change the setting configurations during run time. The only place to assign values to is inside the settings file. :

```
from django.conf import settings

settings.DEBUG = True # Don't do this!
```

Listing 3.13: Wrong import from setting file.

The setting file can contain many sensitive information about the application such as database username and passwords, email addresses, etc. So it is best if we limit access to it by other users. This is very important in a shared-hosting environment.

If we don't want to use the `DJANGO_SETTINGS_MODULE`, we can configure settings manually using following function:

```
from django.conf import settings

settings.configure(DEBUG=True)
```

Listing 3.14: Configure settings manually.

We can pass as many configurations as we want. If a value for a default variable is not set, Django will use the default values in the `global_settings` file. Also, if we have our variables defined somewhere else we can pass the `default_settings` argument in the configure functions:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

Listing 3.15: Configure settings manually.

We should note that if the `DJANGO_SETTINGS_MODULE` is not being used, the call to `configure()` must be made at any point before using any parts of the code, otherwise we will get a `ImportError` exception.

3.7.1 Important configurations

BASE_DIR

Base directory is the build path inside the project. Django uses this to access applications and run them.

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

Listing 3.16: Base Directory.

INSTALLED_APPS

Every time we add a new application to the project, it should be added to this list of installed applications

```
INSTALLED_APPS = [
    'django.contrib.admin',
```

```
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'home',
    'calculator',
    'graphic',
]
```

Listing 3.17: Installed Apps.

MIDDLEWARE_CLASSES

All the middle-ware classes that are installed are declared here:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Listing 3.18: Middle-ware classes.

TEMPLATES

Here we define the configuration for the templates:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ["templates"],
        'APP_DIRS': True,
        'OPTIONS': {
```

```
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Listing 3.19: Templates

DATABASES

We setup the configuration of the databases that the project use such as username, password, ports etc in this list.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Listing 3.20: Database

STATICFILES_DIRS

Here we define the path to the static files such as `HTML`, `CSS` or `JavaScript` files.

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]
```

Listing 3.21: Static files

3.8 URL Dispatchers

The URL configuration for Django applications are set in the `url.py` file. This file has a Python module called `URLconf` which is mapping URLs to their corresponding Python View functions that

is responsible to handle the requests.

Once the user sends a request or tries to access a resource by a URL address, the framework follows a number of steps to determine which function is responsible to take the request. Here is a general overview of the algorithm:

- The Django framework determines which url file should be used. This path to the url file is usually set to the ROOT_URLCONF variable in the settings file.
- Django then will search and loads the variable `urlpatterns`. This variable is a list of `django.conf.urls.url()` instances that maps the URLs to their functions.
- Django will go through all the URL patterns and determines the matched URL.
- Once the matched URL is found, Django will call its View function and will send the `HttpRequest` object and any other arguments that are included in the request pattern.
- If none of the patterns matches the given URL, Django will invoke an appropriate error view.

Here is an example of the `URLconf` module:

```
1 from django.conf.urls import url
2 from . import views
3
4 urlpatterns = [
5     url(r'^index/2003/$', views.index),
6     url(r'^comments/([0-9]{4})/$', views.comments),
7     url(r'^archives/([0-9]{4})/([0-9]{2})/$', views.archive),
8     url(r'^details/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.details),
9 ]
```

Listing 3.22: URL Patterns

The first value of the URL instance is the pattern and the second value is the corresponding view function. For example, in the first instance, URL object is mapping the pattern "*index*/2003" to the view function named `index`. So, every request with the pattern will be sent to the index view function.

Chapter 4

Test Driven Django

4.1 Introduction

Testing web applications is generally a more complex task than other types of programs and applications. There are several layers of logic involved in any web applications such as handling HTTP level requests, processing and validating forms, processing templates, database interactions, etc. But these constraints should not limit the importance and extremely useful role that automated testing plays in web applications. Automated testing benefits the web application in a variety of ways and helps us to solve or avoid a number of problems:

- Making sure the application is working as expected before deployment.
- When adding new code to the application we can use testing to make sure the new functionalists do not change the behavior of application in unexpected way.
- Finding and fixing bugs.
- Testing the performance of the application under heavy loads. etc.

There are also different types of testing that can be performed on web applications such as:

- **Unit testing**: focuses on small units of codes such as methods to verify their correct functionality.
- **Functional testing**: is used to test all the links to the applications such as database connection, form submission, testing sessions, and cookie.
- **Performance testing**: These tests are designed to track the performance and behavior of application under loads. Two of the most general tests are load testing and stress testing.
- **Security testing**: Testing Secure Socket Layer `SSL` connections, input validations, logging and sessions, database injection, etc, are some of the use cases of this testing.

Django has a built-in testing framework with a variety of utilities that can be used to simulate requests, producing and inserting test data, investigating applications output and generally verifying the behavior of the application. Django uses the built-in `unittest` library. there are many different frameworks that are available to test Django applications.

In this chapter, we will describe the testing framework of the Django and will perform `Unit` and `Functional` tests.

4.2 Writing Unit Tests

Django uses the standard python library module unittest. The tests are written using a class-based approach. Every test class is a subclass of `django.test.TestCase` which itself is a subclass of `unittest.TestCase`. Here is an example of testing models. (We will discuss Models in the following sections):

```
from django.test import TestCase
from myapp.models import Patient

class PatientTestCase(TestCase):
    def setUp(self):
        Patient.objects.create(name="Jack", type="A")
        Patient.objects.create(name="Emily", type="B")

    def test_patient_type(self):
        """Identify the Patient's Type."""
        Jack = Patient.objects.get(name="Jack")
        Emily = Patient.objects.get(name="Emily")
        self.assertEqual(Jack.type, "A")
        self.assertEqual(Emily.type, "B")
```

Listing 4.1: TestCase

In the above example, the class `PatientTestCase` will be the test case for the Patient Model. As we can see this class is a subclass of the `django.test.TestCase`. The setup method is used to do any setup work that needs to be done before running the tests such as populating the test database, creating objects and instances of those tables. Once we have the setup we can define all the unit test functions with any method that its name starts with `"test_"`. In the above example the test is called `test_patient_type`.

When we want to run our tests, Django will will search for any file name that starts with `"test"` in all directories. It will automatically build a test suite out of all the test cases and will run the test suit.

When we first build the application, Django will create a default `test.py` file to write test cases for the application. We can restructure this and create a test folder and put test cases for different purposes inside different test files such as `test_model.py`, `test_views.py` and `test_forms.py`.

4.3 Running Tests

We can run test cases using the `manage.py` utility class with the following command:

```
./manage.py test
```

Listing 4.2: Running Tests

The command will find all files named `test*.py`. We can also specify the particular test cases that we want to run by providing any number of tests labels after the previous command. The labels can be a python path to packages, `TestCases`, modules or test methods. For example:

```
# Run all the tests in the patient.tests module
./manage.py test patient.tests

# Run all the tests found within the 'patient' package
./manage.py test patient

# Run just one test case
./manage.py test patient.tests.PatientTestCase

# Run just one test method
./manage.py test patient.tests.PatientTestCase.test_patient_can_speak

#discover tests below that directory:
./manage.py test patient/

#specify a custom filename pattern match
./manage.py test --pattern="tests_*.py"
```

Listing 4.3: Test Options

If at any point we want to stop running the tests, we can send the signal `Ctrl-C`. This will make the test runner to wait and pass the current running test and exit. After exit, the test runner will print the details of all tests that has been passed or failed. If we want to halt the test runner immediately we can use Ctrl-C again , but this time the test runner will not print any details about the tests before the exit.

4.4 The Test Database

In case any of the test cases need access to the database for creating and executing queries, Django will create separate test databases for each test case by mimicking the real database and will destroy them once the test cases are finished, whether test cases pass or fail. This will prevent the tests to use the real production database and accidentally changing the data.

If the SQLite database is used, Django will create an in-memory database by default and will bypass the files system entirely which leads to fast execution of test cases.

An important note is that in case of using databases the test case must be a subclass of `django.test.TestCase` rather than `unittest.TestCase`. The Django TestCases will provide test databases while unittest.TestCase classes avoids running test cases in transaction and flushing databases.

In order to starts all TestCases with a new and clean database, Django will reorder all the test cases in the following manner:

- Every test class that is a subclass of `TestCase` will be run first.
- All the other Django based tests such as `SimpleTestCase` will be run without any particular order.
- Finally any `unittest.TestCase` class that may alter the database and don't restore it to the original stat will be run.

4.5 Test Outputs

When the test runner starts executing tests, it will output some details in the console. We can control the level of details shown with the `verbosity` option in the command line. The following report shows that the database tables are being created before executing tests:

```
1 Creating test database...
2 Creating table myapp_Patient
3 Creating table myapp_Hospital
```

Listing 4.4: Test Database

If all the test cases pass the we should see an message like this:

```
1 ----------------------------------------------------------------------
```

```
Ran 22 tests in 0.221s

OK
```

Listing 4.5: Tests were Successful

And in case any of the messages have failure we will see a detailed report about the failures:

```
======================================================================
FAIL: test_was_published_recently_with_future_poll (polls.tests.
    PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/dev/mysite/polls/tests.py", line 16, in
    test_was_published_recently_with_future_poll
    self.assertIs(future_poll.was_published_recently(), False)
AssertionError: True is not False
----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=1)
```

Listing 4.6: Test Failure Report

In the following sections we will define Models, Views, Forms and Templates, and we will see how we can test each one of these entities of the Django application.

4.6 Models

4.6.1 Defining Models

Django models contains the fields and behavior of the data that we want to store in the database [Moda]. It contains all the essential information that we need about our data. Generally, a model is mapped to a single table in the database. Every model is a python class which inherits the `django.db.models.Model` class and every attribute in this class represents a field in the table.

Here is an example of a table named `Patient` and the model the represents it in as a python class:

```
from django.db import models

class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Listing 4.7: Patient Model

The variables `first_name` and `last_name` are fields of the model. These fields will be mapped to database columns in the `Person` table. The previous model is equivalent to the following `SQL` script:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Listing 4.8: Patient Model

Django will automatically and by default assigns the app name to the table name in the form of `appName_tableName`. Also, an Id field will be generated for the table.

Once we have the model we need to make sure that Django is aware of it by adding our application name to the `INSTALLED_APPS` in the settings file. After adding the application name, we need to run the following command so that Django synchronizes our database with new status of our models.

```
manage.py migrate
```

Listing 4.9: Migrate Command

Whenever we make a change to our models such as adding fields, changing names or deleting models, we need to use migrations. Django will scan models and will apply the changes to the database schemes.

Some of the most common commands are defined below:

- **migrate**: This command is responsible for applying and unapplying migrations.
- **makemigrations**: Creates new migrations based on the changes you have made to your models.

- **sqlmigrate**: Displays the `SQL` statements for a migration.
- **showmigrations**: Lists a project's migrations and their status.

Django also comes with an Admin application [App] that gives us a visual interface to the tables that we have defined in the `model.py` file. In order to use this interface, we have to register our models inside the `admin.py` file as well. The following example will register the `Patient`'s table inside the admin file.

```
1 from django.contrib import admin
2 from .models import Patient
3
4 admin.site.register(Patient)
```

Listing 4.10: Registering in Admin inteface.

Fields are the most important aspects of any model. Any field should be an instance of the Field class. These Field classes define the data type that we want to assign to database columns such as `INTEGER`, `VARCHAR`, `TEXT`, etc. Each one of these fields also takes some optional arguments. For example, we can use the `max_length` argument to limit the number of characters a `VARCHAR` field must contain. Here is some of the important arguments:

- **primary_key**: If this argument is true then the field will be the primary key for the table.
- **default**: This value will provide the default value for the field.
- **unique**: If true, then the field must be unique in the entire table.

Also, Django models allow us to define relationships among tables. The most common type of table relationships are: `many-to-one`, `many-to-many` and `one-to-one`. The following example shows the `many_t_one` relationship between `Manufacturer` and Car assuming that every `Car` has one `Manufacturer` but any `Manufacturer` may produce many different `Car`s:

```
1 from django.db import models
2
3 class Manufacturer(models.Model):
4     # ...
5     pass
6
```

```
class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

Listing 4.11: Many to One relationship

In case we want to add some meta data to our table such as ordering options, table name, etc. we can use the inner class named `Meta` inside the model. For example, in the following model we have set ordering of the table based on the patient's doctor name.

```
from django.db import models

class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    doctor = models.CharField(max_length=30)

    class Meta:
        ordering = ["doctor"]
```

Listing 4.12: Patient Model

Aside from meta data we can also add functions to our models. A common use case is adding the `__str__` fucntion to specify which one of the fields will be used to be shown as the model's title in the interface. In the following example, the Patient's last name will be used as the title for any of the Patient's instances.

```
from django.db import models

class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    doctor = models.CharField(max_length=30)

def __str__(self):
        return self.last_name
```

Listing 4.13: Patient Model Title

4.6.2 Testing Models

Now that we know the definition of Django Models, we will see how we can test these models. In the Test-Driven approach, we first have to write the test cases before implementing the actual code [Modb]. So, in following example we are writing a test case for a model called `Patient` and we want to check if the title of its instances are the same as the Patient's name. We will put the test cases for the Models inside the `test_model.py` file.

We can create an instance of the `Patient` table the same way we create objects from any standard python class. We can pass the fields as arguments to the newly created object. In the following example, we create the instance and assign it to the "Jones" variable. Once we have the object we can access the fields and all the methods that have been defined inside the model.

```
1 from django.test import TestCase
2 from .models import Entry
3
4 class PatientModelTest(TestCase):
5
6     def test_string_representation(self):
7         Adam_Jones = Patient(first_name="Adam", last_name="Jones")
8         self.assertEqual(str(Adam_Jones), Adam_Jones.last_name)
```

Listing 4.14: Patient Model Test

Now, if we remove the `__str__` function from our model and run the test with the `"python manage.py test myapp"` command, we will get the following error:

```
1 Creating test database for alias 'default'...
2 ======================================================================
3 FAIL: test_string_representation (myapp.tests.PatientModelTest)
4 ----------------------------------------------------------------------
5 Traceback (most recent call last):
6 ...
7 AssertionError: 'Patient object' != 'Jones'
8 - Patient object
9 + Jones
10 ----------------------------------------------------------------------
11 Ran 1 test in 0.002s
```

```
12
13 FAILED (failures=1)
14 Destroying test database for alias 'default'...
```

Listing 4.15: Patient Model Test Fail

Here we see that the test runner is creating a test database and the it is trying to run the `test_string_representation` test case. The test is failing and we see that the root of the failure is an `AssertionError`, since the `Patient` object, which will be the default name for every instance of a table in case we don't implement the `__str__` method, is not the same as "Jones", which is the patient's last name, as we expected.

Now if we add the `__str__` method back to our model, and run the test again we see the following report:

```
1 Creating test database for alias 'default'...
2 ----------------------------------------------------------------------
3 Ran 1 test in 0.000s
4
5 OK
6 Destroying test database for alias 'default'...
```

Listing 4.16: Patient Model Test Pass

In the above report, we see that we get an `OK` report which indicates that the test cases have been run successfully and the data base is being destroyed.

4.7 Views

4.7.1 Defining Views

View functions are python functions that as their argument, take web requests and return web responses [Viea]. The type of responses that a view function may return can be `HTML` contents, redirection to other pages, `HTTP` errors, image, `XML` document, etc. Any logic that we have for the application will be used in these functions. Usually any view function maps to a `URL` defined in the `URL` dispatcher. We will have our view functions implemented inside the `views.py` file but it can be restructures based on our needs.

Here is a view function that takes a request and returns the current date and time as an HTML document:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Listing 4.17: current_time View function.

Here is the step by step explanation of the above example:

- First, from the `django.http` module we import the `HttpResponse` class, and python's standard `datetime` library.
- Next, we have defined a view function called `current_time`. The first parameter of view functions is usually the `HttpRequesst` object.
- Finally, the view will return an `HttpResponse` object containing the data and time.

Now in order to show the response at a particular URL, we must add it to the `URL` dispatcher in the `url.py` file. Here we add the view function to the url file:

```
from django.conf.urls import url

from myapp import views

# We are adding a URL called /time
urlpatterns = [
    url(r'^myapp/time/', views.current_datetime, name='time'),
]
```

Listing 4.18: current_time View function URL.

In the above setting we are telling Django to direct any request to the "*http* : *//localhost* : 8000/*myapp*/*time*" to the `current_datetime` view function. The function will receive the response and will return the `HTML` back to be shown in the browser.

Now if we run the server with `"python manage.py runserver"` command and go to the "*/myapp/time*" `URL`, we should be able to see the following response in the browser:

```
It is now 2017-07-30 21:06:55.695828.
```

Listing 4.19: current_time View result.

We notice that the HTML response is generated inside the method. The HTML code will usually be decoupled from the view functionality and will be inside the template folder in the root directory. The view then will return the results to the HTML template and the template then substitutes the results with the variables inside the template and will render the response. To do this we first write the HTML content inside the `time.html` file inside the template folder at the root directory.

```
<html>
    <body>
        It is now {{ time }}.
    </body>
</html>
```

Listing 4.20: time.html Template.

We can use double curly braces inside the HTML files to represent variables. Then we have to send the values that was generated inside the view function as a dictionary to the view template. For this purpose, we can use the `render` function from `django.shortcuts` module. The render function takes the request as the first argument, the path to the template as the second argument and the dictionary of values as the third argument. Here is the view code:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render(request, 'myapp/time.html', {'time' : now})
```

Listing 4.21: current_time View using render function.

Now if we go the previous `URL`, "*http* : *//localhost* : 8000*/myapp/time*", we should be able to see the result:

```
It is now July 30, 2017, 9:10 p.m..
```

Listing 4.22: Time result.

4.7.2 Testing Views

In order to test View function, we need to create `HTTP` requests and send them to these functions. Django provides a test client class which is a python class that acts as web browser and allows us to make and send requests to the views. We can simulate HTTP `GET` and POST requests on the URLs that are defined inside `url.py` files and get the view responses and perform validation checks on them. It also allows us to check if the request is being rendered by the particular Django template that we have defined with the certain values that we have passed to it. [Vieb]

As we have shown earlier in the test-driven approach we first write the test cases and then developer the actual code. Now assuming that we want to have a `current_time` function and we did not implement the view functionality and we don't have the URL pattern in the url.py file, we can write the following test case:

```
1 from django.test import TestCase
2 from django.test import Client
3
4 class MyAppViewTests(TestCase):
5     def test_current_time(self):
6         client = Client()
7         response = client.get('/myapp/time/')
8         self.assertEqual(response.status_code, 200)
```

Listing 4.23: current_time test case.

In the example above, we are importing the Client class from the `django.test` module. In our test case named `test_current_time` we use an instance of the `Client` class to make a GET request to the "*/myapp/time*" URL by passing the URL as the argument to the get function. Here we should note that we do not use the complete URL addresses such as "*http* : *//localhost/example*". The reason is that the test client does not require the web server to be running in order to perform the tests. It will avoid the HTTP overheads and work with the framework directly. This will help the unit tests to be run faster.

We put the test inside `test_view.py` file and will run the test using the `"python manage.py test"` command. We should get the following report:

```
1 Creating test database for alias 'default'...
```

```
System check identified no issues (0 silenced).
======================================================================
FAIL: test_current_time (log.tests.MyAppViewTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\tests.py", line 10, in test_current_time
    self.assertEqual(response.status_code, 200)
AssertionError: 404 != 200


----------------------------------------------------------------------
Ran 1 tests in 0.063s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Listing 4.24: current_time test failure.

We notice that the test has failed. The failure is due to an `AssertionError`. The status code that is returned by the response is 404 which is different from the 202 code that we expect. The reason is that the test is not able to find any URL path matching the "*/myapp/time/*" URL that we are requesting. Now we can add the URL pattern to the `url.py` file as we did earlier and run the test again. Here is the new report:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
======================================================================
FAIL: test_current_time (log.tests.MyAppViewTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\tests.py", line 10, in test_current_time
    self.assertEqual(response.status_code, 200)
AssertionError: 404 != 200


----------------------------------------------------------------------
Ran 1 tests in 0.063s

FAILED (failures=1)
```

```
Destroying test database for alias 'default'...
```

Listing 4.25: current_time test failure.

As we see, the test also fails but for a different reason. The test was able to find the URL path that matches the requested URL. But it was not able to find the `current_time` attribute or function inside the views file in the `myapp` application. The failure is correct since we still have to implement the view function inside the `view.py` file. Now after adding the view function back again like we did previously and run the test again we should get the following result:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
======================================================================
FAIL: test_current_time (log.tests.MyAppViewTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\tests.py", line 10, in test_current_time
    self.assertEqual(response.status_code, 200)
AssertionError: 404 != 200


----------------------------------------------------------------------
Ran 1 tests in 0.063s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Listing 4.26: current_time test successful.

We get an `OK` report indicating the test was able to find the URL, identify the `current_time` view function, make a GET request to it and receives a HTTP response with 200 status code.

4.8 Forms

4.8.1 Defining Forms

Django web framework provides a set of tools and libraries that are helpful to build forms that can be used to accept input from users, process them and respond to those inputs [Fora]. In usual HTML files a form is a collection of `HTML` elements inside a `<form>` tag that allows the user to have

various forms of interactions with the application, such as entering text or selecting options and then send those data back to the server. Generally, any form must be able to specify two things:

- The `URL` that will be receiving the user's input.
- The method that is used to send the data.

GET and POST are two of the most used HTTP methods in web applications forms. These two methods are used for different purposes. Usually, any time a request might be used to change a state of the application or the data in the database, the POST method is used. In contrast GET method is only used when a request is only sent to access some resource in the application on the server. However, one exception would be sending passwords while trying to log in to the application. If we send the password through the GET method, it will be appeared in the URL and eventually in browser history and server logs since GET method uses plain text to send requests. In this case POST method coupled with some of the Django's protection mechanism like `CSRF` protection must be used to transfer sensitive data.

Generally, Django framework handles three parts involved in any form:

- Preparing the data and make it ready for rendering.
- Generating HTML forms for data.
- Processing the input data from users.

Django's `Form` class is responsible for much the work involved in handling forms. It will describe the forms and they it must work and appear in the application the same way the Model class defines objects, their behavior and states. In a similar way that a model object is mapped to a table and its fields to the columns of the table, every form's field is mapped to an `HTML <input>` element. All the fields in the forms have their own classes. These classes are responsible to manage the data and perform any type of validation that is required for that field when the form is submitted. For example `FileField` and `DateField` are two very different type of field that have their own validation process.

The usual processes to render any object are:

- Instantiate it in the view, populate the form with data if required.
- Pass it to the template context.

- Expand the form object to `HTML` markup and populate template variables with the data.
- Receive that posted data by the user.

Building Forms

The following example shows how we can create a simple form to capture the user name as input from the user using simple HTML tags:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name
    }}">
    <input type="submit" value="OK">
</form>
```

Listing 4.27: HTML form.

The above HTML code tells the browser to return the captured data in the form to the "*/your − name/*" `URL` which is defined in the action attribute of the form tag, using POST method, that is defined in the method attribute. There is an input tag inside the form that is used to capture the user input and save it to the `current_name field`.

The above example is a very simple form. However, in most web applications there are tens or even hundreds of forms and fields and many validation processes that is required before processing the data. We can use Django framework Form classes to do most of the work which also help us to decouple the form from the interface code.

Usually we put forms in the `form.py` files inside the application. Here we define the previous form using python `Form` class:

```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

Listing 4.28: Django form.

The above `NameForm` class defines a single field named `your_name` having a label and a `max_length` attribute. The attributes that we define in fields are also used for some types validation. For example, the `max_length` attribute that has value of 100 will check the length of the input to be less or equal to 100 characters. The form also has an `is_valid()` method is runs validation process on all the fields of the form. when this method is called, if all the fields of the form contain data, it will return True and will put the data present in the form inside the `cleaned_data` attribute. The attributes will be generated as attributes inside the input element in the HTML file and templates. The above code will generate the following HTML tags for us:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required />
```

Listing 4.29: HTML tags.

Here we should note that it will not generate the `<form>` tag for us, we have to provide it in the following section.

Once we have defined our form, we have to define the view function that is going to instantiate them form and show it to the user and also receive the posted data by the user. Here is the code to instantiate the form and to receive it:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')
```

```
17
18      # if a GET (or any other method) we'll create a blank form
19      else:
20          form = NameForm()
21
22      return render(request, 'name.html', {'form': form})
```

Listing 4.30: View for Form.

Once the `get_name` view function receives a request, it goes through the if condition. If the incoming request is a `GET` method, it will simply instantiate a `NameForm()` and will return it to the template as a field in the dictionary. We expect this condition the first time the user sends requests to get the page. Once the users fill in the form and press the submit button, the `get_name` view function will be called again, but this time the request is using the `POST` method to carry the input. At this point the first condition is true and we instantiated the `NameForm()` again but we also populate the form with the posted data in the form. Then form's `is_valid()` method is called to check the validity of the data. If the method returns false we go back to the template but with the populated data, which allows the user to correct the input if required. In case the valid method returns true we can access the data in the `cleaned_data` attribute and do any form of processing that is required, such as saving the data to database.

Now that we have the class and the view to instantiate and receive it back we can use the form field returned to the template and place it inside an HTML form tag like the following example:

```
1 <form action="/your-name/" method="post">
2     {% csrf_token %}
3     {{ form }}
4     <input type="submit" value="Submit" />
5 </form>
```

Listing 4.31: HTML using Django Form.

We see that the form is placed inside the form like any other variable that is returned to the template context in a dictionary.

4.8.2 Testing Forms

Let's assume we want to have a form where users can enter their first and last name and submit the form. At this point we only want to have a form named `Patient` with two fields to capture first

and last names. Here we will write the test for the form first and save it inside the `test_forms.py` file inside the tests folder.

```
from django.test import TestCase
from myapp.forms import PatientForm

class PatientFormTestCase(TestCase):

    def test_patient_form_not_null(self):
        patientForm = PatientForm(data={'firstName':"david", 'lastName':"jonse
    "})
        self.assertNotNull(patientForm)
```

Listing 4.32: Testing PatientForm.

As we can see, we want to have a form named `PatientForm` which has two fields named `firstName` and `lastName`. We first have to import the class from the forms file in the application. Once we have the form available we can create an object of the form. We can also initialize the values for the fields in the form by passing a variable named `data`, which is a dictionary containing the data, as an argument to the form's constructor. Django will set the fields in form to the values in the dictionary that matches the fields' names [Forb] .

Now if we try to run the above test case we must expect to get the following error:

```

Creating test database for alias 'default'...
System check identified no issues (0 silenced).
E
======================================================================
ERROR: test_forms (unittest.loader._FailedTest)
----------------------------------------------------------------------
ImportError: Failed to import test module: test_forms
Traceback (most recent call last):
  File "c:\test_forms.py", line 2, in <module>
    from calculator.forms import PatientForm
ImportError: cannot import name 'PatientForm'


```

```
15 ----------------------------------------------------------------------
16 Ran 1 test in 0.000s
17
18 FAILED (errors=1)
19 Destroying test database for alias 'default'...
```

Listing 4.33: Error PatientForm.

The error given is telling us that there are no forms named `PatientForm`. That is because we still have not defined our form in the `form.py` file.

Here we define our form:

```
1 from django import forms
2
3 class PatientForm(forms.Form):
4
5     firstName = forms.CharField()
6     lastName = forms.CharField()
```

Listing 4.34: Defining PatientForm.

The `PatientForm` is inheriting the Django's `Form` class and is having two fields named `firstName` and `lastName` as its fields. Now if we run the test case we get the following report:

```
1 Creating test database for alias 'default'...
2 System check identified no issues (0 silenced).
3 david
4 ..
5 ----------------------------------------------------------------------
6 Ran 2 tests in 0.001s
7
8 OK
9 Destroying test database for alias 'default'...
10 ----------
```

Listing 4.35: Success PatientForm.

So far, we only tested to see if the form is available and we can create an object of it. Next, we want to see if the values that have been sent as argument to the form are valid and are set to

the fields in the form. To test for validity we use the `is_valid()` method of the form which must return true if the data is valid and we can access the field's values of the form by accessing the `data` field which is a dictionary and use the `get` method of the dictionary to access any fields of the form. Here we add two more test, one for testing validity of the data and the other one to make sure the values are set correctly.

Here is the final test case:

```
from django.test import TestCase
from calculator.forms import PatientForm

class PatientFormTestCase(TestCase):

    def setUp(self):
        self.assertTrue("TrueTest")

    def test_patient_form_not_null(self):
        patientForm = PatientForm(data={'firstName':"david", 'lastName':"jones"})
        self.assertIsNotNone(patientForm)

    def test_patient_form_valid(self):
        patientForm = PatientForm(data={'firstName':"david", 'lastName':"jones"})
        self.assertTrue(patientForm.is_valid())

    def test_patient_form_values(self):
        patientForm = PatientForm(data={'firstName':"david", 'lastName':"jones"})
        self.assertEqual(patientForm.data.get('firstName'), "david")
        self.assertEqual(patientForm.data.get('lastName'), "jones")
```

Listing 4.36: Testing PatientForm.

We will run the test case with the following commands:

```
python manage.py test calculator.tests.test_forms.PatientFormTestCase
```

Listing 4.37: Testing PatientForm Command.

We expect to get this report:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
Destroying test database for alias 'default'...
```

Listing 4.38: Final Success PatientForm.

Chapter 5

Implementation

5.0.1 Introduction

In this chapter, we will introduce our design and implementation of the web application based on the tools and components of the Django framework that we have defined in the previous chapters. First, we will review the two major approaches that we had to design a web application that contains machine learning models. These different approaches fundamentally change the scale and the complexity of the web applications. Second, we will overview the structure of our Django project and the applications the we have built. Finally, we will go through different components of our web application and their implementations. We should note that our main focus in this chapter is to review the implementations and different components of the web application. Therefore, we will not cover the test-and-fail phases of our implementation as we did in the examples of chapter 4. The test cases will be provided in the source code.

5.0.2 Design Approach

Based on our introduction to machine learning models in chapter 2, there are two major phases involved in constructing a machine learning model. First, the models need to be trained by the given data that is provided as input. These data would be labeled in case of supervised learning models and unlabeled for unsupervised learning models. In any case, most of these models require different amounts of data, depending on the model, to reach a certain level of accuracy that is desired. In the second phase, after the model is trained and ready to predict, new and unseen data will be given to the model as input to make predictions about the target or independent variable.

In order to build a web application that will host machine learning models, there are three questions that we need to ask and try to answer. These answers will help us to have a better understanding of the scale and complexity of the application.

- How the model will be trained?
- How to persist and re-use the trained model?

Depending on the model and the amount of data it receives for training, the time complexity for the training phase vary significantly. Also, there are more steps involved before feeding the data to the models such as data cleansing. We need to make sure that the data we provide to the model has certain level of accuracy and precision to prevent constructing models with wrong and

misleading attributes. Therefore, the main question is whether we want to train the model inside the application or separate from the application?

In any case, once we have a trained model, the second question is how will we be able to use this model in our web application for prediction, given the unseen data by the clients. The trained models have acquired the state of a set of attributes that need to be persisted in the application, so that we can use it to predict the unseen data. The nature of the state of these models and their attributes also vary. For example, in case of large Neural Networks, we may have to persist the state of tens or hundreds of the network nodes in memory [Che09], while a linear or logistic regression model may only require a few attributes.

In this thesis, we have trained a Logistic Regression model outside of the web application by the training data set. The data set has been cleaned before the training phase and the accuracy of the model has been calculated by the testing data. The following 15 attributes had the most predictive power to predict 5-year survivability of breast cancer incidents. The coefficient of each variable has been given in the second column of the following table.

Variable	Coefficient
race	0.0004
maritalStatus	-0.0305
behaviorCode	-0.9735
grade	0.0186
vitalStatusRecord	-4.6421
histologicType	0.0009
csExtension	-0.0091
csLymphNode	-0.0012
radiation	-0.0175
SEERHistoricStageA	-0.7942
ageAtDiagnosis	0.0274
csTumorSize	-0.0010
regionalNodesPositive	-0.0066
regionalNodesExamined	-0.0001
survivalMonths	1

Table 5.1: 15 Attributes of Logistic Regression Model.

Therefore, in order to persist this trained logistic regression model, we store the coefficient

values inside a data base table. Once we get an input from the client, we will retrieve these values and will run the calculations.

5.0.3 Project Structure

We have divided our project into three main application to decouple different responsibilities across the project. The following are the three main applications:

- **Home**: This application provides the basic home view of the project. Every request to the website will be directed to this application first. Then, clients will be given the options of other applications to choose from. It can also be used to add logging functionality to the application.
- **Calculator**: The Calculator will have the functionality to use the machine learning model and to generate predictions.
- **Graphic**: This application is dedicated to generate any visualization content such as Map, Charts and Graphs that might be required by other applications.

The above applications are the backbone of the project, but the project is not necessarily limited to them. Any new functionality can have its own application added to the structure. Given the instructions in the chapter 3, this additions can be done easily.

5.0.4 Implementation

We will start by presenting the most important files that need to be modified or used for implementation. We should note that we only show snippets of these files that have the important implementations parts and not necessarily the entire content of these files.

Settings

The first thing we need to do after installing applications in the project is to add the application names to the `settings.py` file in the project folder. Here we have added our three applications to the `INSTALLED_APPS` list in the file:

```
1 INSTALLED_APPS = [
2     'django.contrib.admin',
3     'django.contrib.auth',
```

```
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'home',
    'calculator',
    'graphic',
]
```

Listing 5.1: List of Installed Apps

The database is by default `django.db.backends.sqlite3` that we will use during development phase.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Listing 5.2: Databse connection

Since our application has static contents such as `CSS`, `Images` and `JavaScript` files, we can set a path to a folder that contains all these files. Later, we can use the `"manage.py collectstatic"` command that will collect all the static files in all of the project's applications and store them inside a folder named `static` and `static_server`. In case we decide to deploy our application to a server, we can copy and past these folders inside the server and set the path to these static contents in the server configuration files. Then the server will be able to serve these static files to the client requests.

```
STATIC_URL = '/static/'

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]

```

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static_server')
```

Listing 5.3: Static file directories.

URL Dispatcher

Now that we have the applications defined, we have to configure the `URL` patterns inside the `urls.py` file so that Django framework can map every incoming request to the appropriate applications based on the URLs. Here we define four patterns inside the `urlpatterns` list in the `urls.py` file inside the project folder:

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('home.urls')),
    url(r'^calculator/', include('calculator.urls')),
    url(r'^graphic/', include('graphic.urls')),
]
```

Listing 5.4: URL patterns.

Django provides an application that acts as an interface to the `sqlite` database for every project by default. This application is called `admin` [App]. The first pattern maps every incoming request that has a URL pattern containing *admin/* to the `admin` application. Later we will use this interface to populate the data base. The second pattern maps the incoming requests that does not have any pattern in the URL, meaning that they are directed to the root of the application, to the `home` application. As we have discussed earlier the `home` application will act as the landing page of the project, so every client needs to go through this application first. So, the incoming URL such as "$http : /localhost : 8000/$" will be directed to `home`.

The third pattern, maps URLs containing "*calculator/*" to the calculator application and finally the "*graphic/*" pattern will be directed to the `graphic` application. Here we should note that each application can have its own `urls.py` file. So once the root project decided to direct a request to a specific application, the framework will load that application's `urls.py` file to specify which view function needs to handle the request.

The following code is the `urlpatterns` of the `Home` application:

```
urlpatterns = [
    url(r'^', views.home, name='home'),
]
```

Listing 5.5: URL patterns of Home application.

URL patterns of the `Graphic` and `Calculator` applications are given below. Notice that the pattern in every application is an extension to the pattern in the project's main `urls.py` file. For example, any request containing the "$calculator/$" pattern will be forwarded to the `urls.py` file in the `calculator` application. Then if the second part of the pattern includes the "$form/$" pattern then the request will be forwarded to the `form` view function. So, the general URL pattern to the `form` view function is "$calculator/form/$".

```
urlpatterns = [
    url(r'^form/', views.form, name='form'),
    url(r'^result/(\d+(?:\.\d+)?)/', views.result, name='result'),
    url(r'^chart/', views.chart, name='chart'),
]
```

Listing 5.6: URL patterns of Calculator application.

```
urlpatterns = [
    url(r'^map/$', views.map, name='map'),
    url(r'^chart/$', views.chart, name='chart'),
]
```

Listing 5.7: URL patterns of Graphic application.

Model

The Machine Learning model that we have trained and discussed in the previous section will be inside the `Calculator` application. Therefore, we need to define the table that stores the logistic regression's coefficients in the `Calculator`'s `model`. Here we have defined a class called `Logistic_Regression_Model` that inherits the `models.Model` class of the Django framework. The model defines 16 attributes which will store float numbers.

```
class Logistic_Regression_Model(models.Model):

  id = models.AutoField(primary_key=True)
```

```
    race = models.FloatField(max_length = 10, null = True)
    maritalStatus = models.FloatField(max_length = 10, null = True)
    histologicType = models.FloatField(max_length = 10, null = True)
    behaviorCode = models.FloatField(max_length = 10, null = True)
    vitalStatusRecord = models.FloatField(max_length = 10, null = True)
    grade = models.FloatField(max_length = 10, null = True)
    radiation = models.FloatField(max_length = 10, null = True)
    ageAtDiognosis = models.FloatField(max_length = 10, null = True)
    csTumorSize = models.FloatField(max_length = 10, null = True)
    regionalNodesPositive = models.FloatField(max_length = 10, null = True)
    regionalNodesExamined = models.FloatField(max_length = 10, null = True)
    seerHistoricStageA = models.FloatField(max_length = 10, null = True)
    csLymphNode = models.FloatField(max_length = 10, null = True)
    csExtension = models.FloatField(max_length = 10, null = True)
    survivalMonths = models.FloatField(max_length = 10, null = True)
    intercept = models.FloatField(max_length = 10, null = True)

    def __str__(self):
        return str(self.id)
```

Listing 5.8: Logistic Regression Model.

As we will see in the view section we will use this model to retrieves logistic regression attributes once we want to run the prediction.

Admin Interface

Now that we have defined our model, we can use the `admin` application interface to populate the `Logistic_Regression_Model` table in the database. The `admin` application is accessible by going to this `URL`: "$localhost : 8000/admin$". After logging to the application, we can see the tables that are defined in our project.

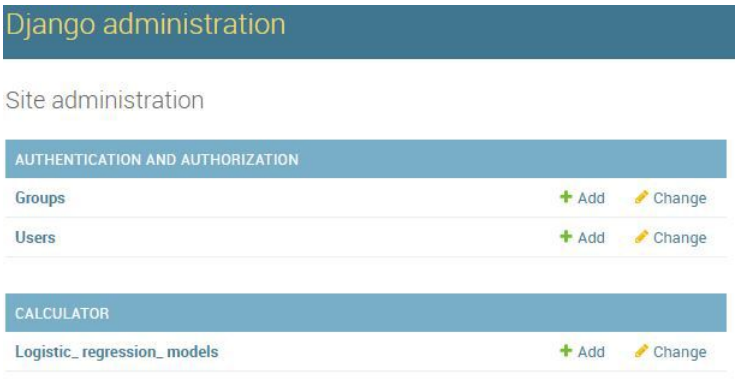

Figure 5.1: Admin Interface

The `Groups` and `Users` tables are created by the Django framework to handle user authentications. We will use the `Logistic_regression_models` table and we will populate it with the coefficient variables define in the table 5.1

Form

Once we have the model defined we need a mechanism to receive the unseen data from the clients and process them. The form that we have defined is called `LogisticRegressionForm` and it is inside the `forms.py` file in the `Calculator` application. Here is a snippet code showing 3 of the total of 16 attributes defined in the form.

```
class LogisticRegressionForm(forms.ModelForm):

  race = forms.ChoiceField(
        widget = forms.Select(
          attrs={
                  'class': 'form-control',
                  'placeholder': '',
                }),

                choices = ([('1','White'), ('2','Black'),]),
                initial='1',
                required = True,
                help_text='100 characters max.'
                )

  histologicType = forms.CharField(
```

```
        widget=forms.TextInput(
                attrs={
                  'class': 'form-control',
                  'placeholder': 'Histologic Type',
                  'type' : 'text'
                }),
        required = True,
        error_messages={'required': 'Please enter Histologic Type.'},
        )

grade = forms.ChoiceField(
        widget = forms.Select(
          attrs={
                  'class': 'form-control',
                  'placeholder': '',
              }),
                choices = ([('1','1'), ('2','2'),('3','3'), ('4','4'),('5','5'
  ), ('6','6'),('7','7'), ('8','8'), ('9','9'),]),
                initial='1',
                required = True,
                )
```

Listing 5.9: Logistic Regression Form.

Django allows us to add `HTML` specific tags to the attributes that are defined in the forms. For example, the `histologicType` attribute is a `CharField` and we have added the `placeholder` tag to it with the value of "*HistologicType*". Therefore, we do not have to add these `HTML` tags to the templates which will further decouple responsibilities. We use this form in our template so that clients can submit the data.

View

The most important view function that we have is inside the `Calculator` application. This view function will receive the client's data through the form that have been submitted and will run the calculations. The function is called `form` and will then return the result to another view function called `result`. The `result` function will display the outcome of the model.

The `form` view function is given below:

```
def form(request):

  if request.method == 'GET':
        # Generate the form
    form = LogisticRegressionForm()
    # Present the form in the form.html template
    return render(request, 'calculator/form.html', {'form' : form, 'result' :
    0})

  elif request.method == 'POST':

        # Populate the form with Client's data.
    form = LogisticRegressionForm(request.POST)
    # Retrieve Coefficients from database.
    modelCoefficients = Logistic_Regression_Model.objects.values().first()
        # Check if the coefficients are present in the model.
    if modelCoefficients != None:
        # check if the form submitted by the client is valid.
      if form.is_valid():
        # Calculate the result
        result = 0
        for i in request.POST:
          if i != 'csrfmiddlewaretoken':
            result = result + (modelCoefficients[i] * int(request.POST.get(i))
    )
        result = result + modelCoefficients["intercept"]
              # Send the outcome to the result view function
        return redirect('result', 10.1)

    return render(request, 'calculator/form.html', {'form' : form, 'result' :
    0})
```

Listing 5.10: form view function.

The `form` view function receives an `HttpRequest` forwarded by the URL configurations that we have defined. The first thing that the function has to determine is whether it has received a `GET`

or POST request. In case the function has received a GET request, it simply return the form.html template which presents the LogisticRegressionForm that we have defined. In the event that the request method is POST, it means that the client has filled in the form and submitted it. Therefore, we catch the data in the POST request and populate the LogisticRegressionForm object. We also retrieve the coefficients that we have stored in the data base using the Logistic_Regression_Model method called objects. This method will return all the rows of the table and we only catch the first row using the first() method. Once we have the coefficients and user's data we run the calculation and store the result in the result variable. We then use the redirect method to send the outcome of the prediction to the result view function. The result view function below simply presents the outcome in the result.html template.

```
1 def result(request, ids):
2
3   return render(request, 'calculator/result.html', {'result' : ids})
```

Listing 5.11: result view function.

The above result view function will receive the ids variable along with the request. The ids variable is the outcome of the prediction that was sent by the form view function. The ids will be passed to the calculator/result.html template in the dictionary.

The view functions of the Home and Graphic applications simply renders the request in their corresponding templates:

```
1 def map(request):
2
3   return render(request, "graphic/map.html")
4
5 def chart(request):
6
7   return render(request, "graphic/chart.html")
8
9 def home(request):
10  return render(request,"home/home.html")
```

Listing 5.12: Home and Graphic view functions.

Templates

The templates that we have defined are all in the `templates` folder in the root directory. Each application has its own folder with the `HTML` files specific to them. The static files such as `CSS`, `Images` and `JavaScript` are inside the `server` folder. In order to use the static files in any template we need to the following code on top of the `HTML` file.

```
{% load staticfiles %}
```

Listing 5.13: Load static files.

Django allows inheritance in templates. So, we have defined a `base.html` template that holds all the tags and information that will be needed by other pages and templates. Therefore, we don't have to write the same code repeatedly. For example, the `map.html` template only defines the tags of the map class inside the `block content`. In order to use `base.html` template, we can use extend the `base.html` template in the `map.html`.

```
{% extends 'base.html' %}

{% block content %}
    <div class="container">
      <div class="row">
        <div class="col-sm-8">
          <div id='map_canvas'></div>
          <p class='pull-right'>
            <div id="map" style="width: 960px; height: 600px"></div>
          </p>
        </div>
      </div>

    </div>
{% endblock %}
```

Listing 5.14: Map template.

In the above code, the `map.html` template has extended the `base.html` template. Once the framework loads this template, it will generate the contents of the `base.html` first and then it will replace the `block content` part of template with the content of the `map.html`.

We have repeated the same pattern in all the other applications which includes the following templates:

- `calculator`
  - `form.html`
  - `result.html`
- `home`
  - `home.html`
- `graphic`
  - `map.html`
  - `chart.html`

Static files

The static files that we have mentioned in the previous sections are all stored in the folder named `static`. We have set the path to this folder in a variable in the `settings.py` file. The most general types of static files in most applications are `CSS`, `Images` and `JavaScript` related libraries. Each application has its own set of static files. For example, the `Graphic` application has `JavaScript` files that are only used to generate Maps and Graphs. Therefore, we don't have to include them in the other applications. The structure of these static folders is given below:

- `calculator`
  - `css`
  - `fonts`
  - `img`
  - `js`
- `home`
  - `css`

- js

- graphic
  - js

In order to create interactive maps by the `Graphic` application we have used an open source `JavaScript` library called `Leaflet` [Prob]. It has a well-documented open source `API` and many plugins that are sufficient for our use case. The `Leaflet` project uses the `OpenStreetMap` [Proc] library and `APIs` to provide the mapping data. The `leaflet-map.js` file uses the `Leaflet` library to construct and configure the map based on the data in the `us-state.js` file.

```
var map = L.map('map').setView([37.8, -96], 4);

var access_token = 'pk.ey';

L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token
    =pk.ey', {
  maxZoom: 18,
  attribution: '<a href="http://openstreetmap.org">',
  id: 'mapbox.streets'
}).addTo(map);
```

Listing 5.15: leaflet-map.js.

To generate charts and graphs we have used another `JavaScript` library called `Highcharts` [Proa]. It provides many different types of interactive and dynamic charts and maps that are heavily used for data visualization. The `highcharts.js` uses this library.

```
Highcharts.chart('container', {
    chart: {
        type: 'scatter',
        zoomType: 'xy'
    },
    title: {
        text: 'Height Versus Weight of 507 Individuals by Gender'
    },
    subtitle: {
```

```
        text: 'Source: Heinz  2003'
    },
    xAxis: {
        title: {
            enabled: true,
            text: 'Height (cm)'
        },
        startOnTick: true,
        endOnTick: true,
        showLastLabel: true
    },
    yAxis: {
        title: {
            text: 'Weight (kg)'
        }
    },
    legend: {
        layout: 'vertical',
        align: 'left',
        verticalAlign: 'top',
        x: 100,
        y: 70,
        floating: true,
        backgroundColor: (Highcharts.theme && Highcharts.theme.
   legendBackgroundColor) || '#FFFFFF',
        borderWidth: 1
    },
});
```

Listing 5.16: highcharts.js.

Chapter 6

Conclusion

In order to build our web application, we needed to choose a programming language and a web developing framework. Since we have extensively used Python to analyze and process our data sets, and also the fact that there are open-source and powerful web developing frameworks based on Python, we have chosen Python and Django for our project.

The Django open source project has many advantages to start developing web applications in a quick and easy way. The framework has a set of tools that are suitable for developing the apps such as Django's developing server and a database. These tools allow us to focus more on developing the web application during developing stages and not to be concerned much about configuring servers and databases.

Moreover, Django framework utilizes the standard Python `unittest` library and has built a powerful testing framework around it. The testing libraries allow us to have a test-driven approach in developing our web application. Here we will show the interface of the web application that we have developed.

6.0.1 Online Survivability Predictor

The online breast cancer survivability predictor web application consists of 2 main parts. The first part uses a form to collect the information regarding the cancer incident and will generate a report showing the result of the prediction based on the model. The second part can be used to generate geographical maps, charts and graphs to provide more information.

The following page is the first page the user will see. It will direct the user to choose one of the two sections of the app.

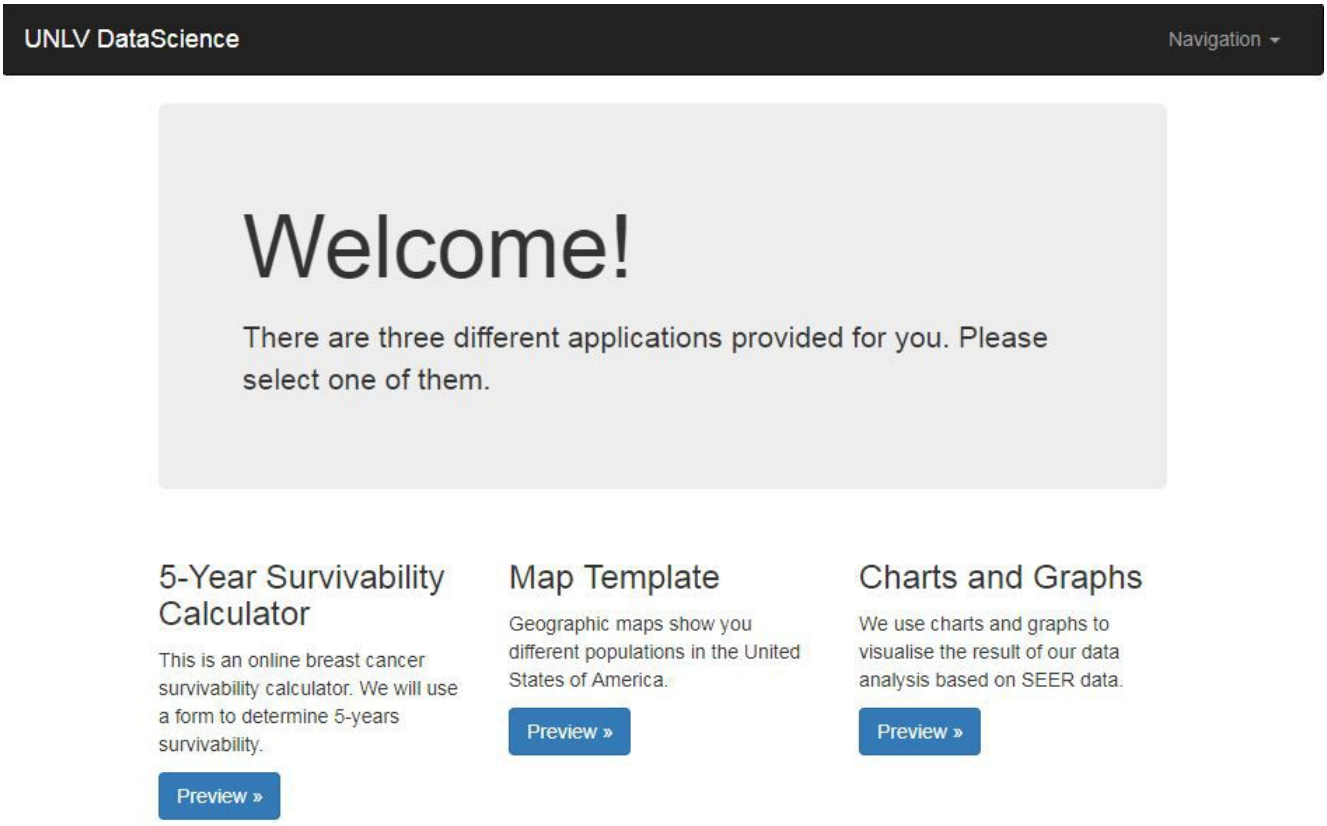

Figure 6.1: First Page.

The online breast cancer survivability predictor web application that we have developed is using 15 attributes that have the most predictive power in the `SEER` data. In order to use the online tool, users need to provide these attributes by filling in the online form provided in the web application.

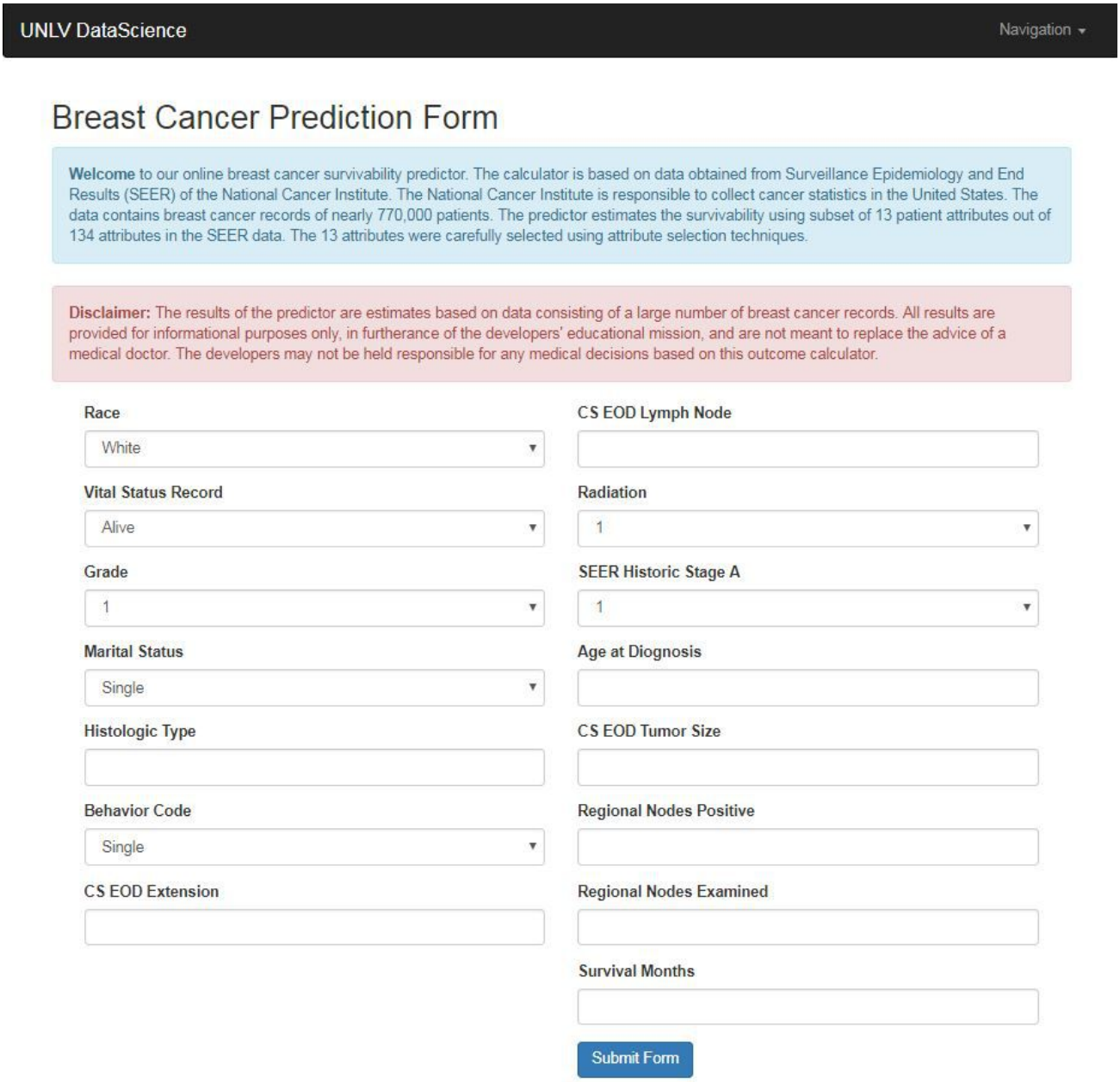

Figure 6.2: Form

Note that all input fields are required to submit the form successfully. After filling in the form and clicking on the "$submitform$" button the data will go through the validation process in the web application and if any of the required fields are empty or if any wrong inputs have been entered (that is violating the type of data specified in the instruction), the user will get an error to correct the data entry. At this point the data will not be submitted until the user modify the input entries and click on the "$submitform$" button again.

After submitting the attributes successfully, the web app will use our machine learning model to process the data and generate the final result. Following the calculations, the user will be redirected to a new page containing the result of the perdition. The outcome of the calculator is shown in the report below.

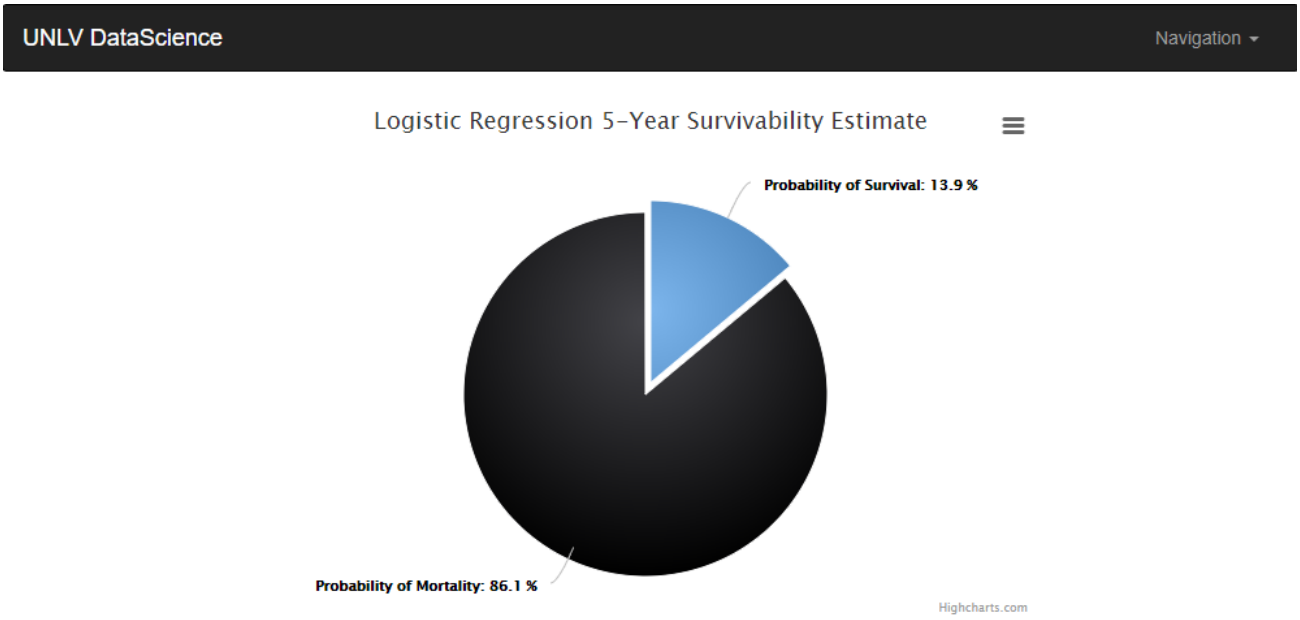

Figure 6.3: Report

6.0.2 Maps, Charts and Graphs

Here we show example of the usage of the Maps, Charts and graphs templates that are in the application and can be used to generate different reports.

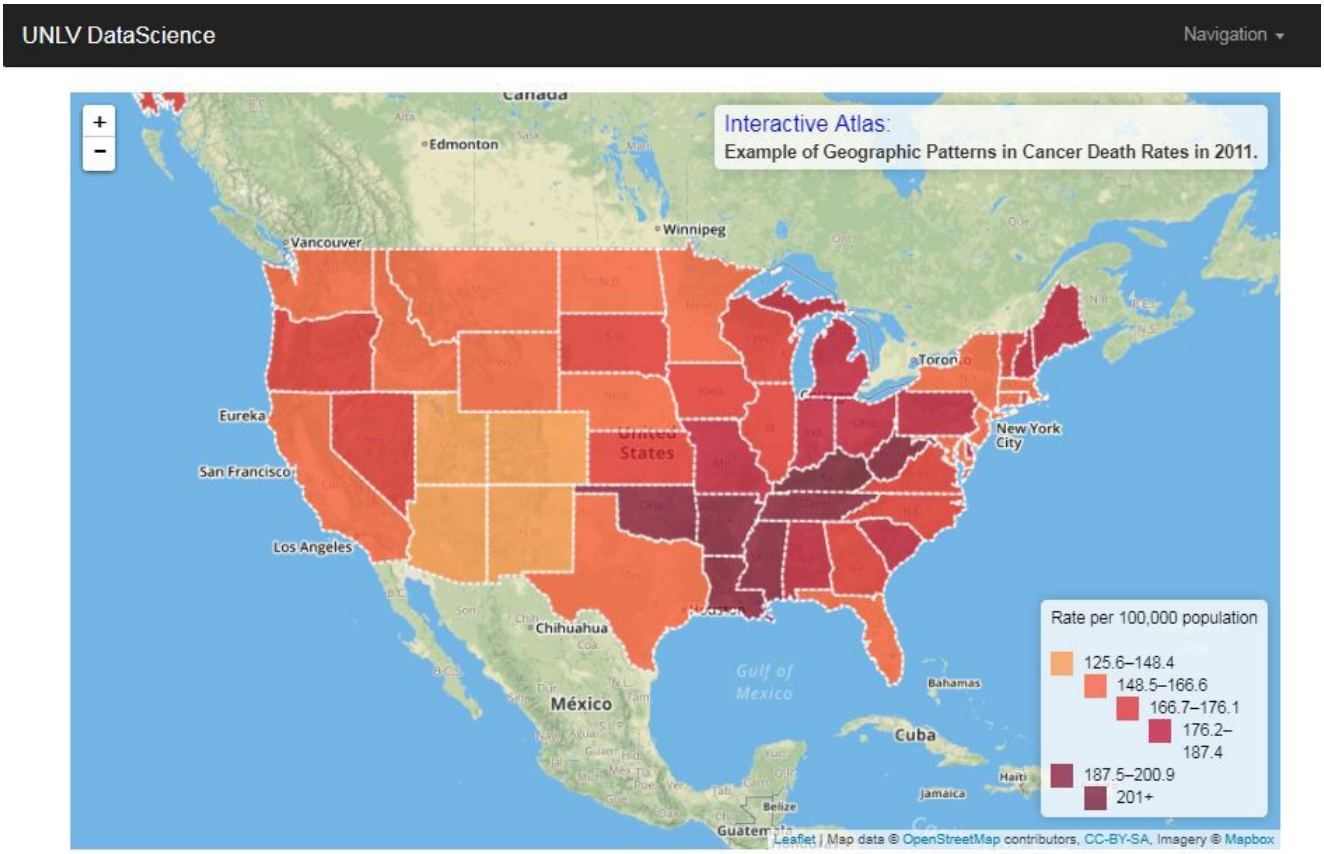

Figure 6.4: Map

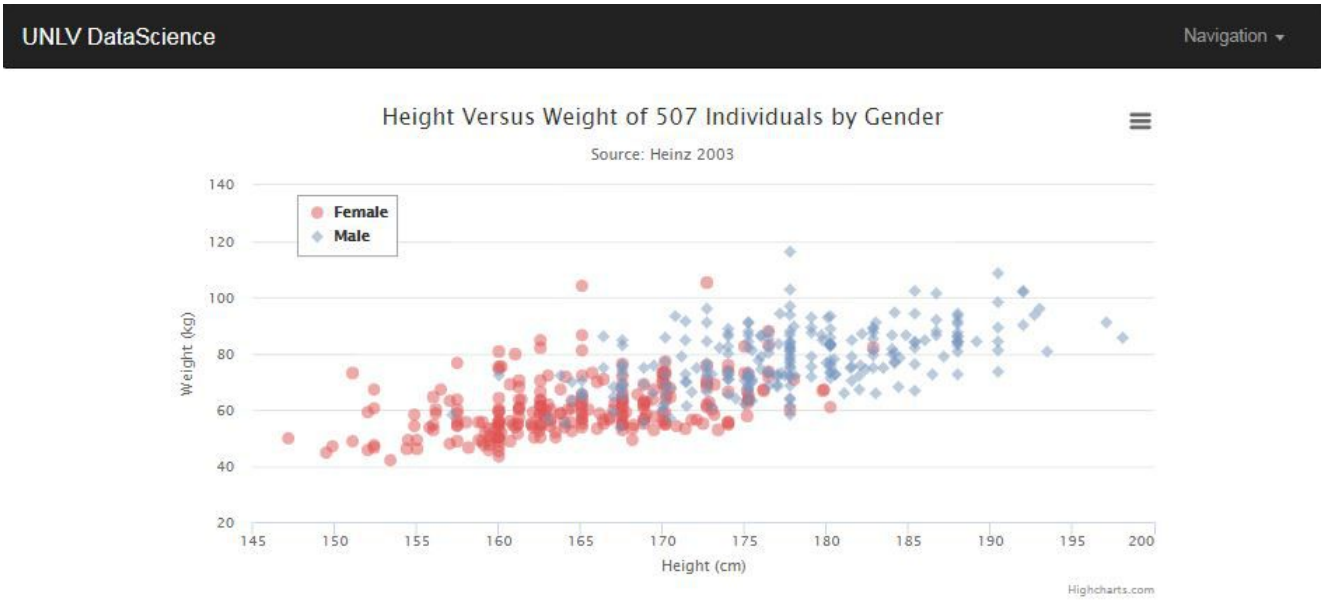

Figure 6.5: Chart

6.0.3 Future Work

So far, we have built a web application in a test-driven development process. The application uses an internal `WSGI` gateway and a SQLite3 database that are built in the Django framework. Once we decide to deploy the application to a production environment the server and database need to be changed to appropriate server and databases that are built specifically for production environments. There are open source servers from Apache and Nginx that can be used for heavy load traffic environments. There are also open source data bases such as MySQL that has more security and many more options that are useful in a production environment.

Another point to mention is the time taken to train a machine learning model and the time it takes for the model to generate a prediction for a given input. Any decision regarding implementing Machine Learning models in web applications need to carefully consider these facts, otherwise the latency for response to an input would be unacceptable. One solution might be to distribute the workload of the model among a cluster of servers. Or another solution might be to generate a model and persist the final model into an in-memory system that can be used to take inputs and generate predicted output. All these solutions require further study and experiments that will be left to future works.

Bibliography

[App] Django Admin Application. https://docs.djangoproject.com/en/1.11/ref/contrib/admin/.

[Bho09] Arthur; Abhishek Bhowmick. A theoretical analysis of Lloyd's algorithm for k-means clustering. 2009.

[Bre84a] J. H.; Olshen R. A.; Stone C. J. Breiman, Leo; Friedman. *Classification and regression trees*. Wadsworth Brooks/Cole Advanced Books Software, 1984.

[Bre84b] J. H.; Olshen R. A.; Stone C. J. Breiman, Leo; Friedman. *Classification and regression trees*. Wadsworth Brooks/Cole Advanced Books Software, 1984.

[Che09] Tianqi Chen. Training Deep Nets with Sublinear Memory Cost. *Cornell University Library.*, 2009.

[Dja] Securing Django. https://docs.djangoproject.com/en/dev/topics/security/.

[Dra98] H. Draper, N.R.; Smith. *Applied Regression Analysis (3rd ed.)*. 1998.

[FC] Django Software Foundation and Contributors. https://docs.djangoproject.com/en/1.11/.

[Fora] Django Forms. https://docs.djangoproject.com/en/1.11/topics/forms/.

[Forb] Test Driven Development Forms. http://test-driven-django-development.readthedocs.io/en/latest/05-forms.html.

[Gai] Decision Trees Information Gain. http://www.cs.cmu.edu/ cga/ai-course/dtree.pdf.

[Mah09] P.; Varadarajan K. Mahajan, M.; Nimbhorkar. *The Planar k-Means Problem is NP-Hard*. 2009.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[Moda] Django Models. https://docs.djangoproject.com/en/1.11/topics/db/models/.

[Modb] Test Driven Development Models. http://test-driven-django-development.readthedocs.io/en/latest/02-models.html.

[oC] Charles Zaiontz Basic Concepts of Correlation. http://www.real-statistics.com/correlation/basic-concepts-correlation/.

[Phi] Django Philosophy. https://docs.djangoproject.com/en/1.7/misc/design-philosophies/.

[Proa] Highcharts Project. https://www.highcharts.com/.

[Prob] Leflet Open Source Project. http://leafletjs.com/.

[Proc] OpenStreetMap Project. http://www.openstreetmap.org/.

[PT12] Andrew Ng Stanford CS221 Artificial Intelligence: Principles and Techniques. http://stanford.edu/ cpiech/cs221/handouts/kmeans.html, 2012.

[Rus03] Peter Russell, Stuart; Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.).* Prentice Hall, 2003.

[Sam59] Arthur Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development.*, 1059.

[Sea67] Hilary L. Seal. *The historical development of the Gauss linear model.* Biometrika, 1967.

[Sp] Epidemiology Surveillance and End Results program. https://seer.cancer.gov/data/.

[Sto] Django Instagram Story. https://engineering.instagram.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad.

[tBPV] Scaling Django to 8 Billion Page Views. https://blog.disqus.com/scaling-django-to-8-billion-page-views.

[Viea] Django Views. https://docs.djangoproject.com/en/1.11/topics/http/views/.

[Vieb] Test Driven Development Views. http://django-testing-docs.readthedocs.io/en/latest/views.html.

[Wis07] Joseph A. Cruz; David S. Wishart. Applications of Machine Learning in Cancer Prediction and Prognosis. *Cancer Informatics*, 2007.

[WSG] Django WSGI. https://docs.djangoproject.com/en/1.11/howto/deployment/wsgi/.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Armin Esmaeilzadeh
armin.esmaeilzadeh@gmail.com

Degrees:

Master of Science in Computer Science 2017
University of Nevada Las Vegas

Thesis Title: A Test Driven Approach to Develop Web-Based Machine Learning Applications.

Thesis Examination Committee:

Chairperson, Dr. Kazem Taghva, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. Ajoy Datta, Ph.D.
Graduate Faculty Representative, Dr. Emma E. Regentova, Ph.D.